НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
“___”_____2020р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Програмне забезпечення розподілених систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Аналітична система вибору кваліфікаційних груп забезпечення дисциплін»

Виконав:
студент IV курсу, групи ТВ-61
Лавренюк Владислав Володимирович _____

Керівник:
старший викладач
Дацюк Оксана Антонівна _____

Рецензент:
кандидат наук, доцент
Сірий Олександр Анатолійович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки: 121 Інженерія програмного забезпечення

Спеціалізація: Програмне забезпечення розподілених систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ

на дипломну роботу студенту

Лавренюку Владиславу Володимировичу
(прізвище, ім’я, по батькові)

1. Тема роботи: Аналітична система вибору кваліфікаційних груп забезпечення дисциплін
керівник роботи: старший викладач Дацюк О. А.
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”25” травня 2020р. № **1168-с**

2. Строк подання студентом роботи: «10» червня 2020 р.

3. Вихідні дані до роботи: мова програмування Kotlin, середовище розробки Android Studio 3.6, менеджер віртуальних пристроїв.

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Розглянути та проаналізувати методи і моделі вибору кваліфікаційних круп, обґрунтувати вибір логічної моделі вибору кваліфікаційних груп та алгоритм розробки мобільного програмного додатку, розробити програмне забезпечення, розробити інтерфейс користувача та забезпечити зручне користування, зробити висновки за результатами роботи.

5. Перелік ілюстративного матеріалу

Діаграма взаємодії компонентів Room, ієрархія представлень, що визначає макет інтерфейсу користувача, діаграма логічних рівнів додатку, архітектура

системи Android, взаємодія та потік даних SQLite, структура колекцій у Cloud Firestore, зв'язки між колекціями та документами Cloud Firestore, схема комунікації Cloud Firestore, редактор макету, скриншоти інтерфейсу користувача.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання "11" жовтня 2019 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи	24.02.2020	
2.	Вивчення та аналіз задачі	12.02.2020– 18.03.2020	
3.	Розробка архітектури та загальної структури системи	27.03.2020– 03.04.2020	
4.	Розробка структур окремих підсистем	09.04.2020– 16.04.2020	
5.	Програмна реалізація системи	17.04.2020– 12.05.2020	
6.	Оформлення пояснювальної записки	03.05.2020– 06.06.2020	
7.	Захист програмного продукту	04.06.2020	
8.	Передзахист	09.06.2020	
9.	Захист	16.06.2020	

Студент _____ Лавренюк В. В.
(підпис) (прізвище та ініціали,)

Керівник роботи _____ Дацюк О. А.
(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

Розроблена система призначена для вибору кваліфікаційних груп викладачів для забезпечення викладання навчальних дисциплін.

Мобільна система виконує підбір списків викладачів для вказаного предмету та проводить аналіз вибору кваліфікаційних груп викладачів.

Побудована інформаційна система використовує сучасні технології для побудови мобільних додатків.

Записка містить 83 сторінки, 32 рисунки, 4 додатки і 9 посилань на електронні ресурси за «Переліком посилань».

Ключові слова: забезпечення дисциплін, кваліфікаційні групи, об’єкт доступу до даних, центральне сховище даних, локальна база даних, Kotlin, Android.

ABSTRACT

The system is designed to select of qualification groups of teachers for provision academic disciplines.

The mobile system performs the selection of information lists of teachers for the specified subject and analyzes the selection of their qualification groups.

The built information system uses modern technologies for building mobile applications.

The note contains 83 pages, 32 figures, 4 appendices and 9 links to electronic resources according to the "List of links".

Keywords: discipline, qualification groups, data access object, central data warehouse, local database, Kotlin, Android.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

1. HDS – інформаційно-довідкова система.
2. Entity (сутність) – представляє таблицю в базі даних.
3. DAO (Data Access Object) – об’єкт доступу до даних.
4. DTO (Data Transfer Object) – об’єкт передачі даних.
5. ORM (Object-Relational Mapping) - об'єктно-реляційне відображення.
6. SDK (Software Development Kit) – набір розробника програмного забезпечення.
7. AVD Manager (Android Virtual Device Manager) – менеджер віртуальних пристроїв Android.
8. IPC (Inter-Process Communication) – механізм міжпроцесорної комунікації.
9. HAL (Hardware Abstraction Layer) – шар абстракції обладнання.
10. RDBMS (Relational Database Management System) – реляційна система керування базами даних.
11. IO (Input-Output) – введення-виведення даних.
12. DB (Database) / БД – база даних.

ВСТУП

Метою роботи була розробка системи формування кваліфікаційних груп викладачів, яка призначена для допомоги в розподілі педагогічного навантаження рівня працівників соціально-педагогічної сфери. Доступність даних є одним із найважливіших факторів для більшості інформаційних та експертних систем. Це дає можливість отримати швидкий доступ до ключових елементів системи, а також у лічені секунди дізнатись про останні зміни, які стосуються конкретної системи в цілому чи окремих її частин, спостерігати динаміку розвитку та змін системи та мати зворотній зв’язок з користувачами та учасниками системи.

HDS (з англ. Help Desk Software, інформаційно-довідкова система) – це програмне забезпечення, що по суті являється автономною сервісною службою, яка включає керування інформацією та (у деяких випадках) керування послугами. Часто обидва терміни вживаються взаємозамінно. Тим не менше, програмне забезпечення довідкової служби спеціально посилається на систему, яка адресує запити користувачів до єдиного головного центру. Такий центр являється впорядкованою сукупністю документів (масивів документів) та інформаційних технологій по збору, обробці, зберіганню та передачі інформації, яка використовується при обробці запиту користувача.

Підприємства та урядові організації використовують інформаційні системи для підвищення ефективності шляхом автоматизації знань та бізнес-процесів, заснованих на документах, та для прийняття більш обґрунтованих рішень через огляди, виявлені у внутрішніх та зовнішніх неструктурованих джерелах інформації.

Програмне забезпечення інформаційної системи різноманітними способами автоматизує обслуговування користувачів. Зазвичай складається з щонайменше трьох частин. До них відносяться керування даними, пакет автоматизації та звітність або оптимізація.

1. ЗАДАЧА РОЗРОБКИ МОБІЛЬНОЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ ДЛЯ ФОРМУВАННЯ КВАЛІФІКАЦІЙНИХ ГРУП ЗАБЕЗПЕЧЕННЯ ВИКЛАДАЧІВ

У наш час все більше і більше навчальних закладів використовують мобільні технології, щоб зробити навчальний процес зручнішим та ефективнішим.

Це дозволяє людині розвиватися і йти в ногу з розвитком мобільних інформаційних технологій. На сьогоднішній день, система Android вважається найпоширенішою мобільною операційною системою, більшість людей користуються саме нею.

Перевагою мобільних інформаційних систем є можливість швидкого доступу до інтернет ресурсів, оскільки ми практично завжди маємо хоча б один такий поряд.

Виходячи з цього, навчальним закладам необхідні системи керування інформацією про студентів, викладачів, навантаження і т.д.

1.1 Призначення

Програмне рішення призначене для методистів та викладачів вищих навчальних закладів, які беруть участь у навчальному процесі. Користувач має можливість миттєво отримати важливу інформацію, зробивши декілька дотиків до сенсорного екрану свого смартфона. Запити можуть бути доступні як в режимі онлайн, так і за відсутності з’єднання, використовуючи кеш, збережений у мобільній базі даних, під час останньої сесії додатку в онлайн режимі.

Таким чином система має можливість отримувати дані з двох джерел: з віддаленої структури керування інформацією та локальної бази даних. Користувач робить запити до локальної структури. Таким чином досягається максимальна швидкість роботи системи. З іншого боку, локальна база даних отримує оновлення, при зміні інформації на сервері. Це дозволяє не тільки підвищити швидкість роботи системи, а також зменшує навантаження на віддалене джерело даних.

1.2 Технічне завдання

1. Вступ

Програмне рішення є аналітичною системою для вибору кваліфікаційних груп забезпечення дисциплін навантаження рівня працівників соціально-педагогічної сфери. Доступність даних є основним факторів та пріоритетом побудови даної системи. Додаток дає можливість отримати швидкий доступ до основних компонентів системи, а також у лічені секунди дізнатись про останні зміни інформації, незалежно від того чи вони стосуються окремих їх компонентів або усієї системи в цілому. Система повинна мати зворотній зв’язок з користувачами та учасниками системи.

1. Призначення

   Програмний продукт в першу чергу призначений для покращення роботи викладачів навчальних закладів. Завдяки розробленому програмному рішенню співробітники, що в першу чергу залучені до навчального процесу, зможуть будь-коли і будь-де отримати потрібну інформацію стосовно навчального процесу та організації роботи персоналу з надання знань.

2. Вимоги до оформлення програмного забезпечення

   Вимоги до оформлення тексту чи позиціонування елементів не мають строгих обмежень. Єдиною важливою вимогою є зручність користування додатком.

3. Цільова аудиторія

   Програмний продукт розрахований в першу чергу на співробітників, які займаються навчанням учнів, не залежно від предмету що викладається, кількості годин чи способу подання матеріалу, будь то читання лекцій чи практичні заняття.

4. Границі проекту

   Цей продукт переслідує чітку ціль для конкретного навчального закладу, а саме підвищення якості підбору персоналу шляхом спрощеного доступу до інформації, необхідної при роботі.

2. Загальний опис

1. Загальний погляд на продукт

Кожного року навчальний заклад повинен забезпечити розподіл навчальних дисциплін згідно наявних викладачів та їх кваліфікацій. Програмний продукт призначений для організації цього процесу. Таке програмне рішення не має аналогів, оскільки слідує цілям конкретного закладу. Проте існує багато схожих за функціями систем для інших, часто досить специфічних напрямків.

2. Особливості продукту

Розроблена система призначена в першу чергу для керування інформацією і має наступні функції:

- доступ та керування списками викладачів;
- доступ до списків навчальних матеріалів для кожної з дисциплін та можливість додавати посилання на нові матеріали з кампусу або інших джерел;
- інформація та керування списками студентських груп;
- доступ до інформації стосовно навчальних дисциплін;
- інформація про посади інструкторів, зареєстрованих в системі;
- доступ та керування кваліфікаціями викладачів;
- інформація стосовно розподілу педагогічного навантаження;
- інформація про користувачів системи;

3. Класи і характеристики користувачів

Всього є 5 ролей: гість, викладач, методист адміністратор та власник системи.

Власник системи має повний доступ до всього функціоналу. Під доступом мається на увазі права читання та запису будь-яких даних. А також може надавати або забирати права адміністратора іншим користувачам.

Адміністратор має майже такий самий доступ до даних, за виключенням перегляду редагування інформації про користувачів (примітка: ніхто не має

можливості переглядати чи змінювати паролі користувачів, навіть власник системи).

Викладач має доступ до основної інформації, яка може знадобитись, у режимі читання.

Гість має найбільш обмежений доступ до даних. Користувач з такою роллю може переглядати лише списки студентських груп, викладачів, навчальних дисциплін та матеріалів.

4. Операційне середовище

   Додаток призначений для мобільних пристроїв, що працюють на платформі Android з версією 7.0 або вище. Для коректної роботи додаток має мати підключення до мережі Інтернет, для того щоб отримувати оновлення інформації в системі та працювати з актуальними даними. Це також необхідно для авторизації в системі.

5. Обмеження дизайну і реалізації

   Оскільки продукт є автономним, тому немає конкретних обмежень для реалізації чи дизайну. Це пов’язано з тим, що немає необхідності підтримки минулих версій. Єдиною вимогою є те, що усі компоненти системи не повинні знімати плату за користування.

3. Функції системи

Розроблений програмний продукт повинен передбачати наступні функції:

- повинна бути побудована система ролей користувачів з їх правами та обмеженнями у інформаційній системі, що описані в пункті 2.3;
- лише користувачі, що мають роль Адміністратор або Власник повинні мати змогу вносити зміни до бази. Винятком являється лише функція додавання навчальних матеріалів, що доступна також і користувачам з роллю Викладач;
- така конфіденційна інформація як посади та кваліфікації викладачів, а також загальний план навантаження повинна бути доступною лише для користувачів що мають роль Адміністратор або вище;

- інформація про список користувачів повинна бути доступною лише для власника системи і баз можливості її редагування. Винятком являється функція зміни ролей для користувачів, що мають роль нижчу ніж поточний користувач (наприклад власник додатку можем максимум наділити іншого користувача правами Адміністратора). Паролі користувачів не повинні бути доступні ні за яких умов;
- уся інформація системи повинна мати централізований доступ. Це може бути віддалений сервер або хмарне сховище;
- дані системи повинні мати локальний кеш, щоб користувач мав змогу переглядати їх навіть за відсутності доступу до мережі;
- додаток повинен бути зручним та простим у користуванні.

2 ОРГАНІЗАЦІЯ РОБОТИ ПІДСИСТЕМ

2.1 Зберігання даних

Для того щоб кожен користувач міг отримати доступ до системи даних, а також робити в ній зміни, які будуть видимі іншим користувачам, потрібна реалізація віддаленого доступу до даних. У якості такого сховища виступає нереляційна база даних Cloud Firestore.

Це база даних документів NoSQL, яка дозволяє легко зберігати, синхронізувати та запитувати дані для мобільних та веб-додатків. Дані можна легко структурувати за допомогою колекцій та документів. Створення ієрархій дає змогу зберігати пов’язані між собою дані та легко отримувати потрібну інформацію за допомогою виразних запитів.

Таблиці баз даних забезпечують найбільш базовий рівень структури даних в базі даних. Кожна база даних може містити кілька таблиць, і кожна таблиця призначена для зберігання інформації певного типу. Кожній таблиці в базі даних присвоюється ім'я, яке повинно бути унікальним у межах конкретної бази даних.

Cloud Firestore легко інтегрується з мобільними та веб-пакетами SDK та вичерпним набором правил безпеки, що дає змогу отримати доступ до бази даних, не потребуючи встановлення окремого серверу. Використовуючи хмарні функції, серверний обчислювальний продукт, може виконати розміщений серверний код, який реагує на зміни даних у базі. Це дає змогу створювати справді безсерверні програми.

Це рішення також дає змогу автоматично синхронізувати дані між пристроями. Користувачі можуть отримати доступ до своїх даних та змінити їх у будь-який час, навіть коли вони в режимі офлайн. Це дає можливості для синхронізації даних.

База дає можливість обмежити доступ до даних на основі інформації про користувача, тобто залежно від його відповідності до вказаних шаблонів доступу до даних. Такий функціонал забезпечує високу безпеку на рівні користувачів.

2.2 Локальна база даних

Локальні бази даних отримують інформацію із хмарного сховища даних. Локальне сховище представлено реляційною базою даних SQLite. Для зручності та надійності була використана бібліотека Room, яка являє собою рівень абстракції над SQLite, щоб забезпечити більш надійний доступ до бази даних, використовуючи повну потужність SQLite.

Бібліотека допомагає створити кеш даних розробленої програми на пристрої, на якому працює додаток. Цей кеш служить єдиним джерелом даних програмної системи, дозволяє користувачам переглядати послідовну копію ключової інформації у розробленому додатку, незалежно від того, чи мають користувачі Інтернет.

Оскільки додаток обробляє нетривіальну кількість структурованих даних, це дає змогу отримати велику користь від збереження цих даних локально завдяки кешуванню відповідних фрагментів даних. Таким чином, коли пристрій не може отримати доступ до мережі, користувач може переглядати цей вміст, коли він перебуває в режимі офлайн. Будь-яка зміна даних, ініційована користувачем, синхронізується з сервером після відновлення доступу до мережі.

Оскільки в Room передбачені ці моменти, було прийнято рішення використовувати Room замість SQLite.

У створеному локальному сховищі реалізовані є 3 основні компоненти:

Database: Містить точку входу для бази даних і служить основною точкою доступу для базового з'єднання до збережених реляційних даних створеного додатку.

Entity (сутність): Представляє таблицю в базі даних.

DAO (з англ. Data Access Object, об’єкт доступу до даних): Містить функціонал, необхідний для доступу до бази даних.

Додаток використовує базу даних Room для отримання об'єктів доступу до даних або DAO, пов'язаних з цією базою даних. Потім додаток використовує кожен DAO, щоб отримати об'єкти зі сховища і зберегти будь-які зміни, які вони повертають до бази даних. Нарешті, додаток використовує Entity для отримання та встановлення значень, що відповідають стовпцям таблиць у базі даних.

Взаємозв'язок між різними компонентами Room зображено на рисунку 2.1:

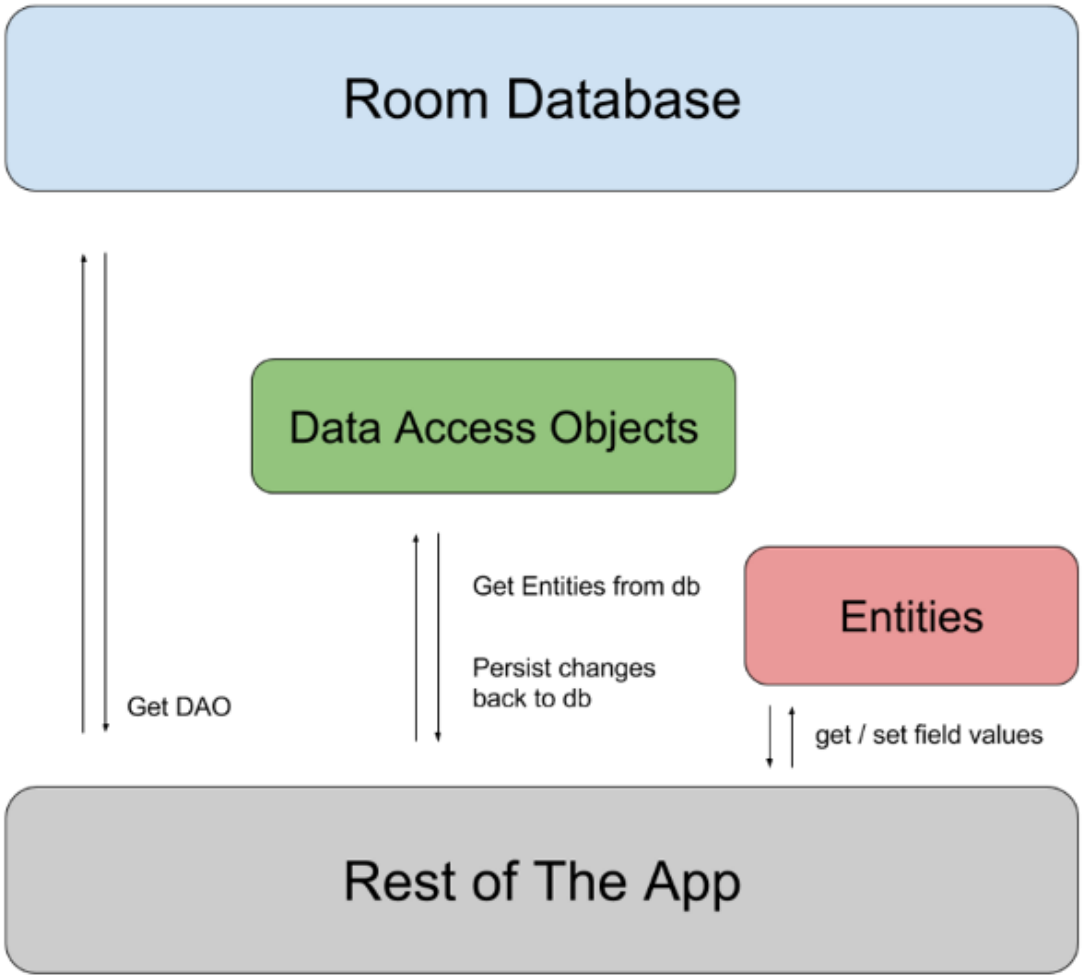


Рисунок 2.1 – Діаграма взаємодії компонентів Room

2.2.1 Функції мобільних баз даних

Мобільні обчислювальні пристрої, включаючи смартфони, персональні цифрові помічники, планшетні комп'ютери та переносні пристрої, використовуються для зберігання даних та обміну ними між мережами. Це дає користувачеві величезні переваги для розповсюдження даних та економії витрат та часу.

Замість того, щоб підтримувати центральну базу даних і споживати сховище даних для спілкування кожного разу, коли їм це потрібно, можна використовувати мобільну базу даних як рентабельне цінне рішення. Мало того, що за допомогою мобільної бази даних ми можемо заощадити додаткові витрати на передачу даних між сервером та мобільними пристроями. Багато додатків вимагають завантажувати дані та інформацію з джерел даних і їм потрібно керувати цією інформацією, навіть коли вони перебувають поза зоною дії та відключені від мережі.

Мобільні бази даних відокремлені від основної бази даних і можуть легко переноситися в різні місця, навіть якщо вони не підключені до основної бази даних. Вони все ще можуть спілкуватися один з одним для обміну даними. Мобільна база даних включає кілька функцій:

- головна системна база даних, яка зберігає всі дані та пов'язана з мобільною базою даних;
- мобільність, дозволяє користувачам переглядати інформацію навіть під час руху;
- пристрій використовує локальну базу даних для доступу до інформації;
- мережевий зв'язок, який дозволяє передавати дані між мобільною базою даних та основною базою даних.

2.2.2 Аналіз локальної бази даних

Більшість часу операційна система Android використовує SQLite як технологію мобільних баз даних. SQLite є реляційною базою даних, яка використовується в різних вбудованих платформах iOS та Android. SQLite - це бібліотека програмування на C і має розмір близько 500 Кбіт. Вона має багато переваг і недоліків. Серед них:
Переваги SQLite:

1. Немає основних залежностей.
2. Можна визначати схему та структуру даних за своїм бажанням.
3. Повний доступ до створення та маніпулювання базою даних.
4. Можуть захопити базу даних та проаналізувати її при тестуванні.
5. Простота у використанні та взаємодії.

Недоліки SQLite

1. Дуже багато практично однакових дій при роботі з даними.
2. Провайдери вмісту - ще один рівень, необхідний для реалізації.
3. Немає процесу перевірки на рівні компіляції.
4. Оновлення схеми вручну (технічне обслуговування, сценарії міграції)
5. SQL – ще одна мова, яку потрібно знати.
6. SQL-запити можуть тривати довше.

2.2.2 Огляд альтернативних систем

Сьогодні існує багато альтернатив SQLite, які були розроблені як рішення для недоліків. Якщо протокол спілкування являється недостатньо зручним, завжди можна використовувати абстракцію об'єкта поверх SQLite. Це зроблено за допомогою ORM (Object Relational Mapping). Однак якщо при бажанні повністю замінити SQLite, можуть бути використані одна із безлічі альтернатив, таких як Couchbase Lite, Interbase, LevelDB, Oracle Berkeley DB, Realm, SnappyDB, Sparksee Mobile, SQL Anywhere, SQL Server Compact і UnQLite. Всі вони мають відмінності за такими характеристиками як тип збереження даних, методи синхронізації, зв’язки між об’єктами, рівень шифрування, бізнес модель, розмір тощо.

2.3 Об’єкти передачі даних

Система має протокол спілкування, який є проміжним шаром між локальною базою даних та хмарним сховищем і описує алгоритм потоку даних між цими структурними компонентами та їх взаємодію. Частиною такого протоколу спілкування є DTO (з англ. Data Transfer Object, об’єкт передачі даних), який відповідає за перетворення даних з однієї структури в іншу та навпаки.

Різниця між об'єктами передачі даних та бізнес-об'єктами або об'єктами доступу до даних полягає в тому, що DTO не має жодної поведінки, крім зберігання, пошуку, серіалізації та десеріалізації власних даних (мутаторів, аксесуарів, аналізаторів та серіалізаторів). Іншими словами, DTO - це прості об'єкти, які не повинні містити ділової логіки, але можуть містити механізми серіалізації та десеріалізації для передачі даних як масиву байтів.

2.4 Користувацький інтерфейс

Користувацький інтерфейс програмного додатку – це все, що користувач може бачити та з чим взаємодіяти. У Android він являє собою ієрархію макетів та віджетів. Макети - це об’єкти ViewGroup (контейнери), які керують тим, як розташовані їхні дочірні представлення на екрані. Віджети – це об'єкти View (представлення), компоненти інтерфейсу, такі як кнопки та текстові поля.

Контейнери визначають структуру компонування для їх представлень та інших контейнерів, як показано на рисунку 2.2

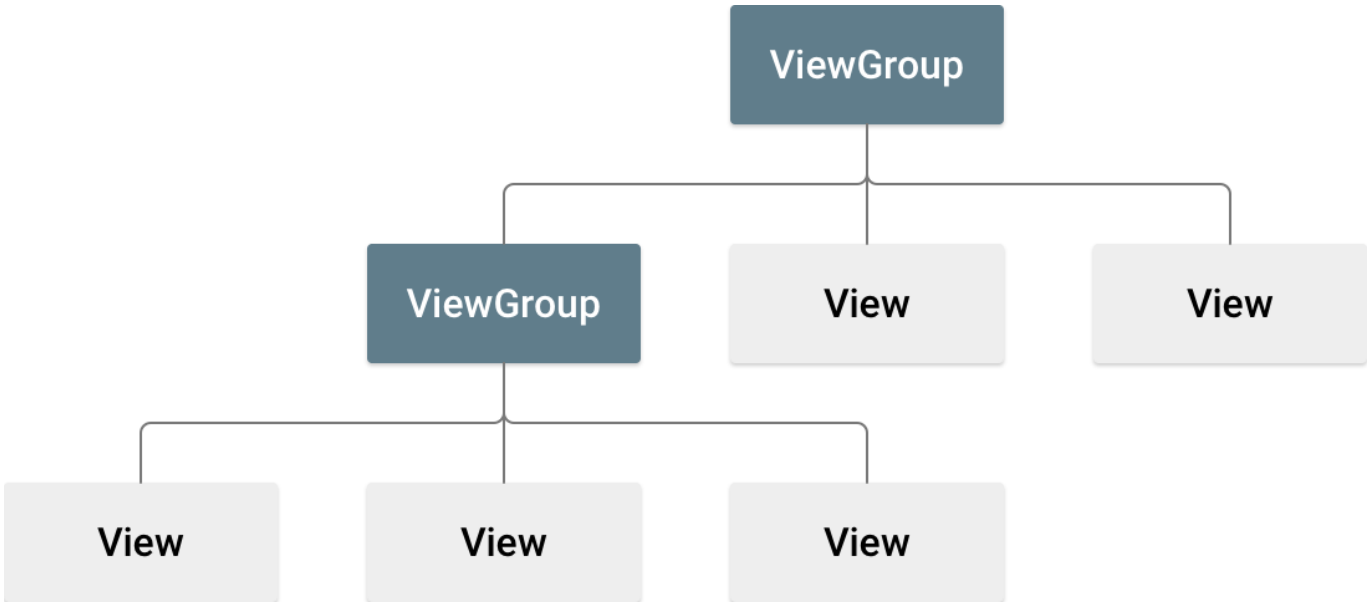


Рисунок 2.2 – Ілюстрація ієрархії представлень, яка визначає макет інтерфейсу користувача

2.5 Взаємодія підсистем

Описані вище підсистеми утворюють рівень даних створюваної програми і представляють доступ у вигляді сховищ даних, які інкапсулюють роботу з вищеописаними джерелами даних та протокол спілкування між ними, представляючи єдиний інтерфейс для читання та запису інформації. Такий підхід забезпечує простоту в роботі там не накладає суттєвих обмежень для подальшого розширення системи.

3. СТРУКТУРА ПРОГРАМНОГО РІШЕННЯ

Основними пріоритетами при створенні програмного рішення були можливість розширення та зручність у користуванні. Беручи це за основу, архітектура додатку була поділена на рівні. В результаті з додатку можна виділити наступні логічно-виокремлені рівні:

- рівень даних,
- рівень сервісів,
- рівень інтерфейсу користувача,
- рівень бізнес логіки.

Метод декомпозиції або розбиття задачі на частини всім відомий і активно використовується майже у кожній сфері. З точки зору розробки програмного забезпечення, такий підхід сприяє кращому розумінню системи, що в свою чергу полегшує розробку і подальший розвиток програмного продукту. Оскільки кожен компонент виконує конкретне завдання і не залежить від інших, то при внесенні змін до одного модуля, не потрібно буде вносити додаткові зміни до компонентів, з якими цей модуль взаємодіє. Схема взаємодії рівнів зображена на рисунку 3.1.

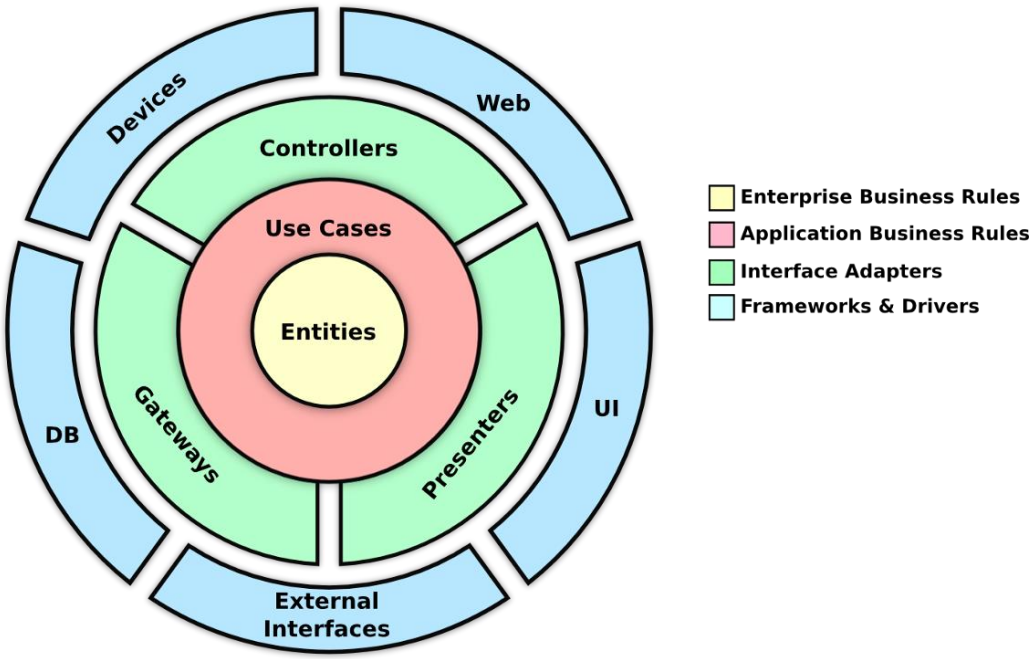


Рисунок 3.1 – Діаграма логічних рівнів додатку

В центрі – рівень даних (жовтий колір). Він взаємодіє лише з бізнес логікою програми (червоний). Далі йде інтерфейс користувача (зелений) і рівень сервісів (синій). Шар бізнес логіки у свою чергу взаємодіє з усіма іншими рівнями в тій чи іншій мірі. Як видно з діаграми, немає чіткого розмежування між трьома зовнішніми шарами. Це пов’язано з тим що останній, рівень сервісів, може взаємодіяти напряму як з рівнем бізнес логіки (наприклад доступ до якогось інтернет ресурсу, представленого сервісом), так і з рівнем інтерфейсу користувача (наприклад камера), в залежності від самого сервісу. А коли, наприклад, потрібно витягнути інформацію з бази даних і зобразити на екрані, запит делегується до відповідного компоненту на рівні бізнес логіки, потім виконуються запити до бази, і отримані дані знову потрапляють на екран користувача через рівень бізнес логіки.

Архітектура програмного додатку була побудована слідуючи архітектурному принципу MVVM. Архітектура Model-View-View-Model не тільки надає значення фрейворку під час створення мобільного додатку на стороні клієнта, але й встановлює основу для інших функцій, таких як прив'язка даних та сфери застосування.

За допомогою архітектури MVC можна ізолювати логіку програми від рівня інтерфейсу та підтримувати розділення проблем. Контролер отримує всі запити на додаток і працює з моделлю, щоб підготувати будь-які дані, необхідні для перегляду. Перегляд використовує дані, підготовлені контролером, і відображає остаточну презентабельну відповідь. [7]

Як випливає з назви шаблону, модель MVVM складається з трьох основних компонентів або частин.

Модель (Model) включає в себе бізнес логіку та дані, які використовуються додатком. У сценарії розробленого програмного продукту модель була абстрагована в різні компоненти. Наприклад, у програмі є сховище, яке відповідає за отримання та збереження об'єктів через сервіс, який відповідає за взаємодію зі сховищем даних.

Фактична реалізація Моделі не важлива для загального шаблону MVVM, хоча реалізація об'єктів у Моделі може впливати на те, як об’єкти будуть викриті в ViewModel. Модель не повинна мати жодної логіки відображення даних.

Модель представлення (ViewModel) інкапсулює логіку відображення та дані, які використовуються представленням даних. Іншими словами, модель представлення виконує роль посередника і між цими компонентами і бере на себе обов’язки керування та взаємодії. ViewModel несе відповідальність за прийняття об'єктів даних, доступних у Моделі, та надання їх доступними як властивості, до яких дані можуть прив'язувати дані.

Відображення (View) - це те, що бачить користувач на екрані - всі елементи інтерфейсу користувача.

3.1 Рівень даних

Як вже було написано раніше, розроблений програмний продукт має два джерела даних (з англ. Data Source): центральну базу даних Cloud Firestore та мобільну базу даних Room, основу якої складає SQLite.

Room є частиною компонентів архітектури Android, представлених в Google I/O 2016. Room не являється лише об'єктно-реляційним відображенням (ORM, з англ. Object-Relational Mapping), це ціла бібліотека, яка дозволяє керувати базаою даних SQLite на іншому рівні. Використовуючи анотації, можна визначити бази даних, її таблиці та операції; Бібліотека автоматично переведе ці анотації в інструкції / запити SQLite для виконання відповідних операцій в двигун БД.

Одним з основних компонентів рівня даних є репозіторії (Repository), що перекладається як сховище даних. Це структури програмної системи, які відповідають за усі необхідні операції між центральною та локальною базами даних, а саме:

- Оновлення локальної бази даних, для зберігання актуальної інформації
- Конвертація об’єктів головної бази даних в об’єкти локальної
- Конвертація сутностей локальної бази даних в структуру центральної
- Взаємодія з рівнем бізнес логіки
- Відправлення та завантаження даних з основного джерела даних
- Вся робота з локальною базою даних

Коли дані у головній базі оновлюються, програма отримає інформацію про це і репозіторій зробить всю необхідну роботу по завантаженню цих даних, збереженню їх у локальній базі, а як наслідок – оновлення інтерфейсу користувача. Якщо ж додаток не був відкритий у цей момент, або доступ до інтернет ресурсів був відсутній, всі ці дії відбудуться при наступному підключенні.

Окрім керування даними репозіторії також представляють свого роду міст рівнем даних та рівнем бізнес логіки програми.

3.2 Рівень бізнес логіки

Бізнес логіка розробленої системи являє собою зв’язок між інтерфейсом користувача та рівнем даних. Саме цей рівень відповідає за обробку кожної дії, що зробив користувач та делегує це іншим компонентам системи і, коли потрібно, повертає відповідний результат користувачу у вигляді інформації.

У даній системі елементи бізнес логіки представлені об’єктами ViewModel, які відповідальні за зберігання та керування даними пов'язаними з інтерфейсом користувача, сприятливим способом життєвого циклу активності. Це дозволяє пережити дані зміни конфігурації, такі як обертання екрана. Деталі взаємодії з користувацьким інтерфейсом такі як активності та конфігурації буде розглянуто в наступному пункті.

Каркас Android управляє життєвими циклами контролерів користувальницького інтерфейсу, такими як діяльність (Activity) та фрагменти (Fragment). Рамка може вирішити знищити або знову створити контролер користувальницького інтерфейсу у відповідь на певні дії користувача або події пристрою.

Якщо система знищує або знову створює контролер інтерфейсу користувача, будь-які перехідні дані, пов’язані з інтерфейсом користувача, які в них зберігались, втрачаються. Наприклад, додаток може включати список користувачів у одній із його діяльності. Коли активність заново створена для зміни конфігурації, нова активність повинна повторно отримати список користувачів.

Інша проблема полягає в тому, що контролерам інтерфейсу користувача часто потрібно здійснювати асинхронні запити, для повернення яких може знадобитися певний час. Контролеру користувальницького інтерфейсу потрібно керувати цими запитами та забезпечувати очищення ресурсів системи після їх знищення, щоб уникнути можливих витоків пам'яті. Це керування вимагає великого обслуговування, і у випадку, коли об’єкт заново створюється для зміни конфігурації, це призводить до марного витрачання ресурсів, оскільки об’єкт, можливо, повинен буде перезапустити всі дії, які він вже зробив.

Контролери інтерфейсу користувача, такі як діяльність та фрагменти, в основному призначені для відображення даних інтерфейсу користувача, реагування на дії користувача або обробки зв’язку операційної системи, наприклад запитів дозволу. Вимагаючи того, щоб контролери інтерфейсу також відповідали за завантаження даних з бази даних або мережі, додають перевантажують логіку відповідальної за це структури і порушують цим самим одне з правил проектування правильної системи – принцип єдиного обов’язку. Призначення надмірної відповідальності контролерам інтерфейсу може призвести до отримання одного класу, який намагається самостійно впоратися з усією роботою програми, замість делегування роботи іншим класам. Призначення таким чином надмірної відповідальності контролерам інтерфейсу також ускладнює тестування.

Для вирішення цієї проблеми були створені архітектурні компоненти ViewModel для контролера інтерфейсу, який відповідає за підготовку даних для інтерфейсу користувача. Об'єкти ViewModel автоматично зберігаються під час зміни конфігурації, щоб дані, які вони зберігають, були негайно доступними для наступної діяльності або її фрагмента. Наприклад, при відображенні списку користувачів у додатку, потрібно переконатися, що відповідальність за придбання та збереження списку користувачів покладена на ViewModel.

3.3 Рівень користувацького інтерфейсу

Інтерфейс програми – це перше, що користувач бачить і взаємодіє з ним. З точки зору користувача, каркас підтримує загальний принцип роботи для кожного додатку, встановленого на смартфоні чи планшеті. У той же час, з точки зору розробника, цей каркас забезпечує деякі основні блоки, які можна використовувати для побудови складного та послідовного інтерфейсу користувача.

Компонент діяльності (Activity) є найважливішим компонентом програми для Android, а спосіб його запуску є основною частиною моделі додатків мобільної платформи. На відміну від парадигм програмування, в яких програми запускаються методом main(), запускає екземпляр Activity, що відповідає певному етапу життєвого циклу програмної системи.

Розробка мобільних додатків відрізняється від робочого столу тим, що взаємодія користувача з додатком не завжди починається в одному місці. Натомість подорож користувачів часто починається недетерміновано. Наприклад, якщо відкрити додаток електронної пошти на головному екрані, можна побачити список електронних листів. На противагу цьому, якщо використовується додаток для соціальних медіа, який потім запускає програму електронної пошти, можна перейти безпосередньо до екрана програми електронної пошти для написання листа.

Ідея Activity була розроблена для полегшення цієї парадигми. Коли один додаток викликає інший, цей додаток викликає активність в іншому додатку, а не додаток як атомне ціле. Таким чином, діяльність служить вхідною точкою для взаємодії програми з користувачем.

Діяльність надає вікно, в якому додаток малює свій інтерфейс користувача. Це вікно, як правило, заповнює екран, але може бути меншим, ніж екран, і плавати вгорі інших вікон. Як правило, одна активність реалізує один екран у додатку. Наприклад, одна з дій програми може реалізувати екран налаштувань, а інша реалізує екран вибору фотографій.

Більшість додатків містять кілька екранів, а це означає, що вони містять кілька дій. Зазвичай одна активність у додатку визначається як основна діяльність, яка є

першим екраном, який з’являється, коли користувач запускає додаток. Кожна діяльність може потім розпочати іншу діяльність для виконання різних дій. Наприклад, основна діяльність у простому додатку електронної пошти може надавати екран, на якому відображається вхідний лист електронної пошти. Звідси основна діяльність може запустити інші види діяльності, які надають екрани для таких завдань, як написання електронної пошти та відкриття окремих електронних листів.

Не менш важливими структурними компонентами користувацького інтерфейсу є фрагменти. Вони представляють поведінку або частину інтерфейсу користувача. Є можливість об'єднати кілька фрагментів в одній діяльності, щоб створити інтерфейс з декількома панелями та повторно використовувати фрагмент у кількох заходах. Фрагмент можна розглядати як модульний розділ діяльності, який має власний життєвий цикл та власні правила поведінки відповідно до дій користувача, які можна змінювати під час виконання діяльності (на кшталт "додаткової діяльності").

Кожен структурний компонент діяльності має свій життєвий цикл які в сукупності формують життєвий цикл всього додатку.

3.4 Рівень сервісів

Цей рівень лише на половину відноситься до програмного продукту, оскільки він відповідає за взаємодію додатку з іншими додатками як частину їх системи або використання інших додатків, систем або їх модулів. Іншими словами, сервісний рівень представляє собою інтеграцію програмного додатку з іншими системами. Прикладом інтеграції сервісу в мобільний додаток є використання камери для того щоб зробити знімок з камери, або використання технології BLE (Bluetooth Less Energy) для позиціонування за допомогою лише функції блютуз і програмного додатку.

У розробленій програмі використовується сервіс для керування користувачами Firebase Authentication. Знання ідентичності користувача дозволяє додатку надійно зберігати дані користувачів у хмарі та надавати однаковий персоналізований досвід на всіх пристроях користувача.

Firebase Authentication надає надійний сервіс для автентифікації користувачів розробленої системи. Він підтримує аутентифікацію, використовуючи електронні адреси та паролі, номери телефонів, надсилаючи SMS-повідомлення на телефони, популярних федеральних постачальників ідентифікаційних даних, таких як Google, Facebook та Twitter.

Аутентифікація Firebase тісно інтегрується з іншими службами Firebase і використовує такі галузеві стандарти, як OAuth 2.0 та OpenID Connect.

Компонент Firebase Auth реалізує кращі практики аутентифікації на мобільних пристроях та веб-сайтах, що дозволяє покращити авторизацію та реєстрацію для програмного додатку. Сервіс також може обробляти граничні випадки, такі як відновлення облікового запису та з'єднання облікових записів, які можуть бути чутливими до безпеки та схильні до помилок.

Щоб авторизувати користувача у додатку, розроблена система спочатку отримує облікові дані автентифікації від користувача. Ці облікові дані можуть бути електронною адресою користувача та паролем. Потім ці дані передаються в SDK (з англ. Software Development Kit, набір розробника програмного забезпечення) для аутентифікації. Потім Firebase Authentication сервіс перевіряє ці дані щоб повернути відповідь клієнту.

Після успішного входу в систему буде надано доступ до основної інформації користувача

3.5 Бібліотека LiveData

У розробленому програмному рішенні була використана сучасна бібліотека для спостереження даних LiveData (з англ. - живі дані). На відміну від звичайних спостережуваних, LiveData усвідомлює життєвий цикл програми, тобто означає, що вона поважає життєвий цикл інших компонентів додатків, таких як діяльність, фрагменти чи сервіси. Це усвідомлення забезпечує, що LiveData оновлює лише спостерігачів компонентів програми, які перебувають у активному стані життєвого циклу. [8]

LiveData вважає спостерігача, представленого класом спостерігача, в активному стані, якщо його життєвий цикл перебуває у стані Started (розпочатий) або Resumed (відновлений). LiveData повідомляє лише активних спостерігачів про оновлення. Неактивні спостерігачі, зареєстровані для перегляду об’єктів LiveData, не отримують сповіщення про зміни.

Ви можете зареєструвати спостерігача в парі з об'єктом. Цей взаємозв'язок дозволяє усунути спостерігача, коли стан відповідного об'єкта життєвого циклу зміниться на Знищений (Destroyed). Це особливо корисно для діяльності та фрагментів, оскільки вони можуть спокійно спостерігати за об’єктами LiveData і не турбуватися про витоки даних - активність та фрагменти миттєво відписуються від оновлень, коли їх життєві цикли знищені. [2]

Використання LiveData забезпечує такі переваги:

- Впевненість, що інтерфейс користувача відповідає актуальному стану даних.

  LiveData відповідає шаблону спостерігачів. LiveData сповіщає об’єкти спостерігача, коли стан життєвого циклу змінюється. Замість оновлення інтерфейсу користувача щоразу, коли дані програми змінюються, спостерігач може оновлювати інтерфейс користувача щоразу, коли є зміни.

- Ніяких витоків пам'яті

  Спостерігачі прив’язані до об'єктів життєвого циклу та очищають їх після знищення пов'язаного з ними життєвого циклу.

- Жодних збоїв через припинену діяльність

  Якщо життєвий цикл спостерігача неактивний, як, наприклад, у випадку коли додаток запущений, але неактивний, перекритий іншим тощо, він не отримує жодних подій LiveData.

- Більше немає ручного керування життєвим циклом

  Компоненти інтерфейсу просто спостерігають за відповідними даними та не припиняють і не продовжують спостереження. LiveData автоматично

керує всім цим, оскільки вона знає про відповідні зміни стану життєвого циклу під час спостереження.

- Завжди актуальні дані

  Якщо життєвий цикл стає неактивним, він отримує останні дані, коли знову стає активним. Наприклад, діяльність, яка була у фоновому режимі, отримує останні дані відразу після повернення на перший план.

- Правильні зміни конфігурації

  Якщо активність або фрагмент відтворено внаслідок зміни конфігурації, наприклад обертання пристрою, він одразу отримує останні наявні дані.

- Обмін ресурсами

  Існує також можливість розширити об’єкт LiveData, використовуючи шаблон Одинак, обернувши системні сервіси так, щоб вони могли бути спільними для всього додатку. Об'єкт LiveData підключається до системної служби один раз, і тоді будь-який спостерігач, який потребує ресурсу, може просто спостерігати за об’єктом LiveData.

Використання LiveData з бібліотекою Room:

Room - бібліотека керування об'єктами бази даних, що спрощує доступ до бази даних в додатках Android.

Замість того, щоб приховувати деталі SQLite, Кімната намагається прийняти їх, надаючи зручні інтерфейси (API) для запитів в бази даних, а також перевіряти такі запити під час компіляції. Це дозволяє отримати доступ до всієї потужності SQLite, зберігаючи при цьому безпеку типів даних, яку забезпечують конструктори запитів.

Бібліотека стійкості Room підтримує спостережувані запити, які повертають об’єкти LiveData. Спостережувані запити записуються як частина об'єкта доступу до бази даних (DAO).

Ця бібліотека генерує весь необхідний код для оновлення об’єкта LiveData при оновленні бази даних. Створений код виконує запит асинхронно на фоновому потоці, коли це необхідно. Цей шаблон корисний для збереження даних, що відображаються в інтерфейсі користувача, синхронізованих з даними, що зберігаються в базі даних.

3.6 Середовище розробки

Створення мобільного додатку відбувалося в програмному середовищі Android Studio 3.6.3.

Android Studio є офіційним інтегрованим середовищем розробки (IDE, англ. Integrated Development Environment) для операційної системи Android від Google, побудованого на основі програмного забезпечення JetBrains IntelliJ IDEA і розробленого спеціально для розробки Android додатків. Це середовище підтримує такі мови для розробки програмного забезпечення як: Kotlin, Java та C++.

Дуже важливим фактором є постійне оновлення та підтримка даного середовища компанією Google, яка покращує його та розширює існуючий функціонал.

Крім потужного редактора коду та інструментів для розробників IntelliJ, Android Studio пропонує ще більше функцій, що підвищують продуктивність при створенні програм для Android, таких як:

- гнучка система побудови проекту на основі Gradle;
- створення варіантів та генерація групи apk-файлів;
- шаблони коду, які допомагають створювати загальні функції програми;
- багатий редактор макетів з підтримкою редагування стилю перетягування;
- інструменти виділення для отримання продуктивності, зручності використання, сумісності версій тощо;
- можливості ProGuard та підписання додатків;
- вбудована підтримка Google Cloud Platform, що спрощує інтеграцію з хмарними сервісами Google;

Інструментарій, який був використаний для створення, тестування, запуску та упаковки програмного продукту є частиною системи збірки Android. Ця система збірки замінює систему Ant, яка використовується із Eclipse ADT. Вона може

працювати як інтегрований інструмент з меню Android Studio та незалежно від командного рядка.

Гнучкість системи збирання Android дозволяє досягти максимальної цілісності, не змінюючи основні файли джерела програми. У розробці та тестуванні програмного продукту переважно використовувались такі елементи як менеджер віртуальних пристроїв так монітор пам’яті.

Менеджер віртуальних пристроїв Android (AVD Manager, з англ. Android Virtual Device) дозволяє створювати віртуальні девайси для швидкого тестування додатку що розробляється. В меню AVD Manager є можливість вибрати найпопулярніші конфігурації пристроїв, розміри екрана та роздільну здатність для попереднього перегляду додатків.

AVD Manager постачається з емуляторами для пристроїв Nexus та Pixel, а також підтримує створення користувацьких наборів зовнішнього вигляду (skins) на основі конкретних властивостей емулятора та привласнення цих скінів до апаратних профілів. Android Studio встановлює прискорювач емуляторів шивдкого виконання програмного забезпечення Intel® x86 Hardware Accelerated Execution (HAXM) і створює емулятор за замовчуванням для швидкого прототипування додатків.

Монітор пам'яті Android Studio забезпечує перегляд стану пам’яті у реальному часі, даючи можливість легше відстежувати використання пам’яті додатку для пошуку розміщених об’єктів, пошуку витоків пам’яті та відстеження кількості пам’яті, що використовується підключеним пристроєм.

4. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Призначення системи

Кожного року, потрібно виконувати розподіл педагогічного навантаження, вручну перебираючи великі об'єми інформації, такі як набір навчальних дисциплін, кваліфікації викладачів, наявність навчальних матеріалів, тощо.

Кваліфікаційні групи відбираються на основі ліцензійних вимог до викладачів, тобто видів і результатів професійної діяльності особи за спеціальністю, яка застосовується до визнання кваліфікації, відповідної спеціальності (рисунок 4.1).

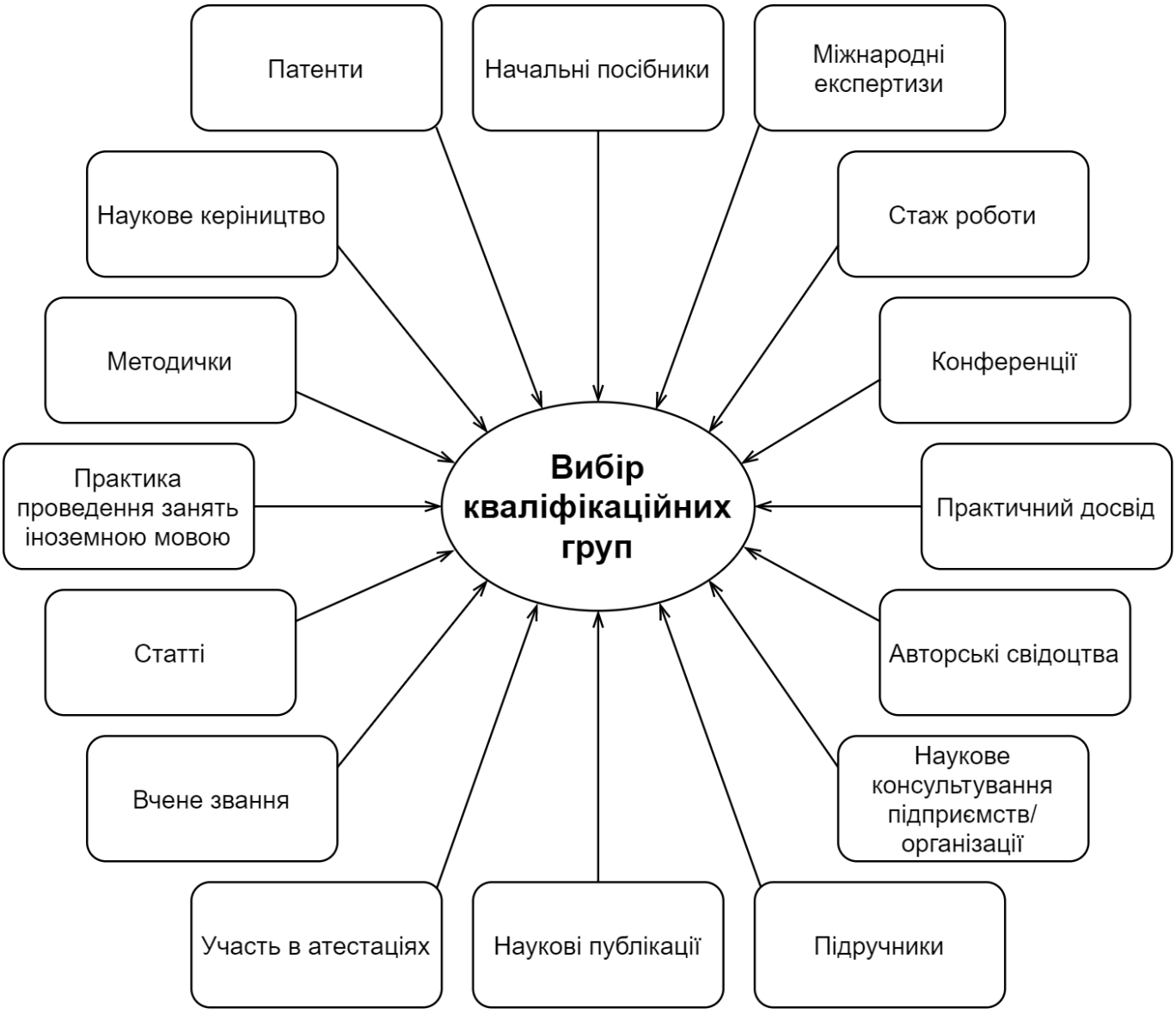


Рисунок 4.1 – Критерії вибору кваліфікаційних груп

Розроблене програмне рішення призначене для того щоб структурувати інформацію, яка може знадобитись для працівникам освітніх установ. А також щоб кожна зацікавлена особа змогла отримати ці дані просто взявши в руки свій смартфон. Цільовою аудиторією є працівники соціально-педагогічної сфери.

Вирішення питання доступності даних дає можливість отримати швидкий доступ до ключових елементів системи, та у лічені секунди дізнатись про останні зміни, які стосуються системи в цілому чи окремих її частин.

Вимоги до реалізації не накладають чітких обмежень, проте дають вказівки про сутність системи та задають напрям її розвитку.

4.2 Архітектура додатку Android

Потік даних у системі відбувається за принципом асинхронного доступу до даних шляхом відправлення запитів (рисунок 4.2).

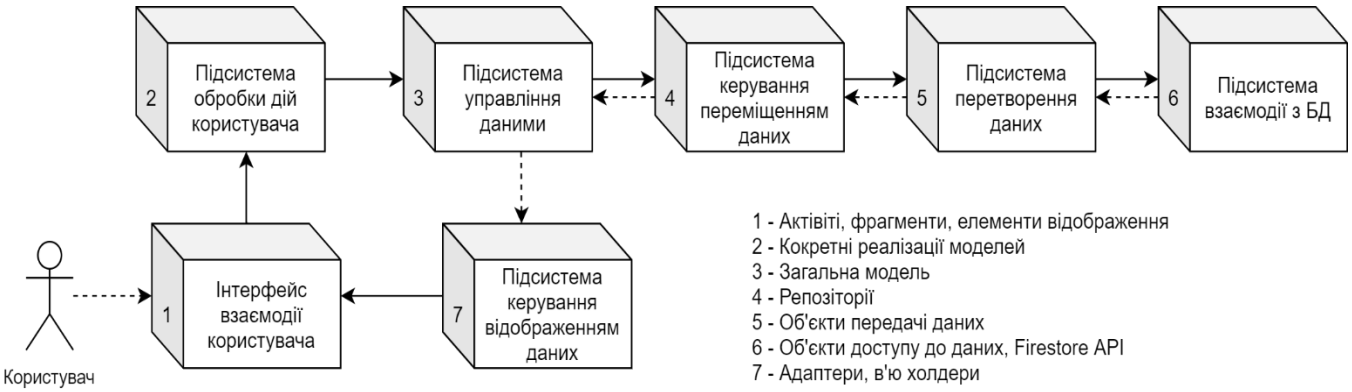


Рисунок 4.2 – Потік даних

Такий підхід дає можливість користувачу взаємодіяти з програмою навіть під час опрацювання запитів, замість того щоб чекати поки інформація не буде отримана.

Таким чином як тільки запит був оброблений, відбувається передача даних, поки не досягне інтерфейсу користувача, де останній зможе побачити доставлену інформацію або її оновлення.

Архітектура розробленої системи Android містить такі компоненти (рис. 4.3):

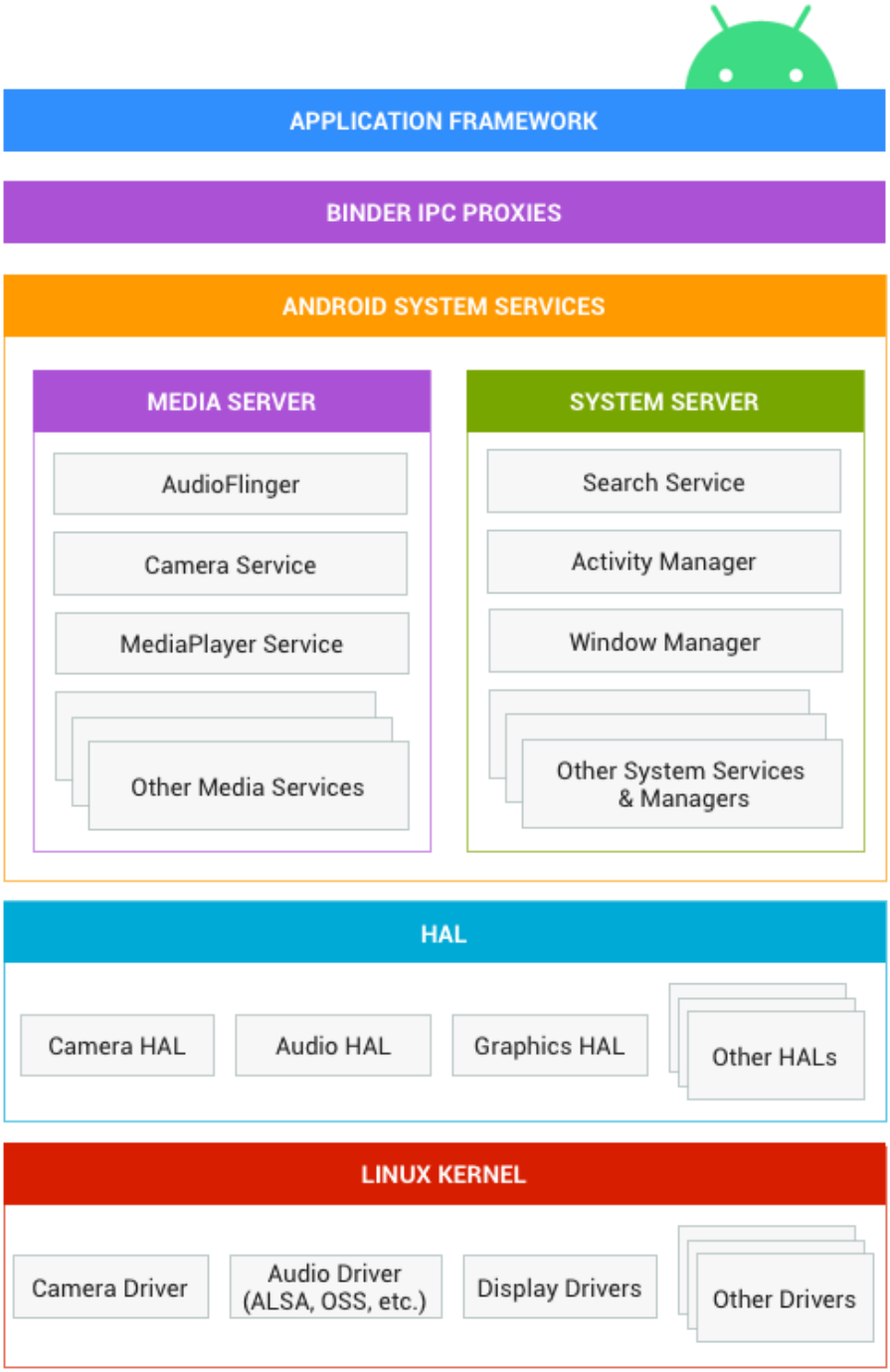


Рисунок 4.3 – Архітектура системи Android

Прикладний каркас або фреймворк (Application Framework) використовується у процесі розробки додатку. API для розробників якомога більше відображається безпосередньо на базових інтерфейсах HAL і може надавати корисну інформацію про впровадження драйверів.

Механізм міжпроцесорної комунікації Binder (IPC, з англ. Inter-Process Communication) дозволяє фреймворку програми об’єднувати межі процесів та

викликати код сервісних систем Android. Це дає можливість API високого рівня взаємодіяти з системними сервісами Android. На рівні фреймворку, цей обмін повідомленнями прихований від розробника.

Системні сервіси (System services) - це модульні, орієнтовані компоненти, такі як менеджер вікон, служба пошуку або диспетчер повідомлень. Функціональність, підпорядковується API фреймворку програми та взаємодіє із системними службами для доступу до базового обладнання. Android включає дві групи сервісів: системні (наприклад, менеджер вікон та диспетчер повідомлень) та медіа (послуги, що беруть участь у відтворенні та записі об’єктів даних).

Шар абстракції обладнання (HAL, з англ. Hardware Abstraction Layer) визначає стандартний інтерфейс для постачальників апаратних засобів для впровадження, що дозволяє Android проявляти агресивність щодо реалізації драйверів нижчого рівня. Використання HAL дозволяє реалізувати функціональність, не впливаючи або змінюючи систему вищого рівня. Реалізації HAL упаковані в модулі та завантажуються системою Android у відповідний час.

Розробка драйверів пристроїв схожа на розробку типового драйвера пристроїв Linux. Android використовує версію ядра Linux (Linus Kernel) з кількома спеціальними доповненнями, такими як Low Killer (система керування пам'яттю, яка є більш агресивною щодо збереження пам’яті), блокування відновлення (системна служба керування енергозберіганням PowerManager), драйвер IPC Binder та інші важливі функції для мобільної вбудованої платформи.

Основна концепція програми, побудованої на асинхронних запитах, завдячує побудованій архітектурі додатку (рисунок 4.4).

Activity представляє собою екран мобільного додатку, з яким користувач може взаємодіяти. Тобто виконувати певну активність, звідси й назва - Activity.

Мобільний Android додаток має свій життєвий цикл. Так, наприклад коли один екран перекривається іншим, більшість дій пов’язані з попереднім, такі як наприклад

анімації, призупиняються. Подібних нюансів насправді дуже багато і їх потрібно враховувати при розробці мобільного додатку.

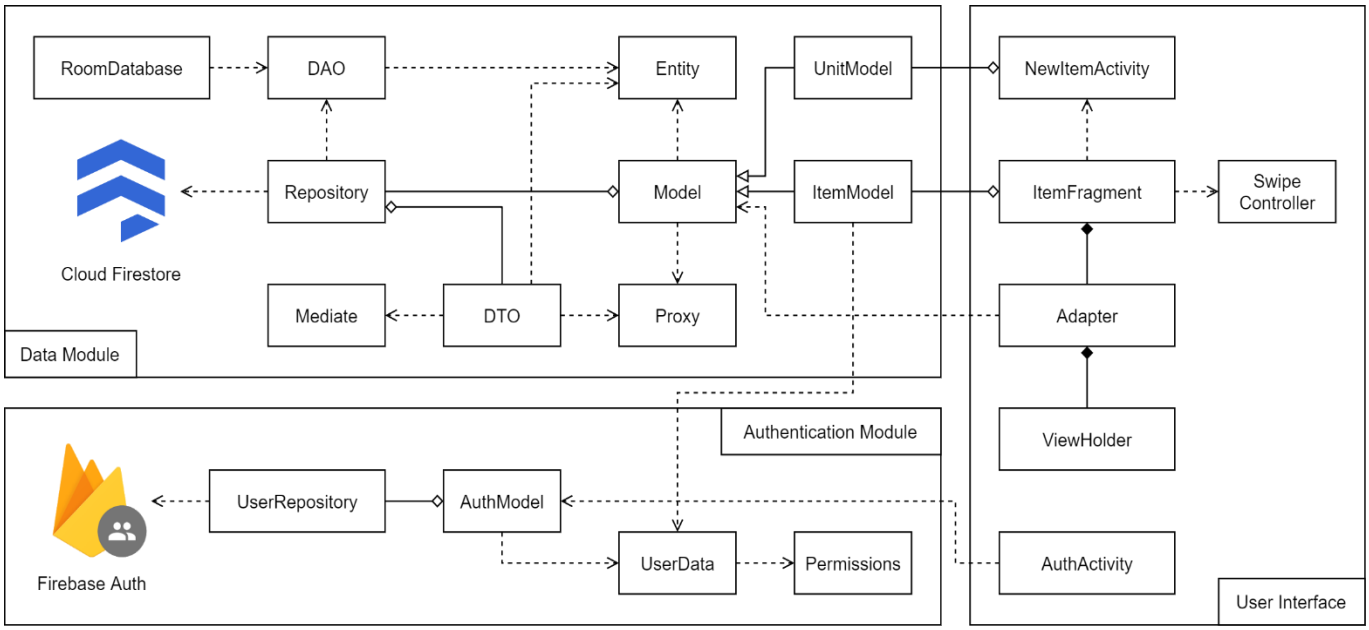


Рисунок 4.4 – Узагальнена діаграма класів

ViewModel є одним із архітектурних компонентів Android і призначений для збереження необхідної інформації про поточну діяльність користувача. ЇЇ перевагою є те, що вона знає про життєвий цикл податку та може виконувати різні дії залежно від поточного стану.

LiveData також є одним із архітектурних компонентів Android і являє собою метод доступу до інформації. При якому будь-які зміни з цими даними, відслідковуються і програма отримує відповідне повідомлення. Це позбавляє нас від постійного звертання до джерела інформації з метою перевірки актуальності даних, що в результаті підвищує продуктивність додатку.

Репозіторії – призначені для роботи з джерелами інформації (тобто базами даних). Таким чином модель просто каже які дані вона хоче отримати, або змінити. А відповідний репозіторій вже потурбується про все інше.

Для відображення даних користувачеві додатку був розроблений так званий адаптер. Такі класи використовуються спеціально для того, щоб відображати список даних. Аналогічно клас Observer, тобто спостерігач, був розроблений, щоб відслідковувати зміну даних в базі, щоб потім вносити відповідні зміни.

Це досягається завдяки компоненту LiveData, яка зберігається в моделі та є джерелом даних для адаптера.

Схема, зображена на рисунку 4.5 відображає взаємодію компонентів системи на прикладі однієї таблиці. Word у даному прикладі означає один елемент списку або рядок у таблиці.

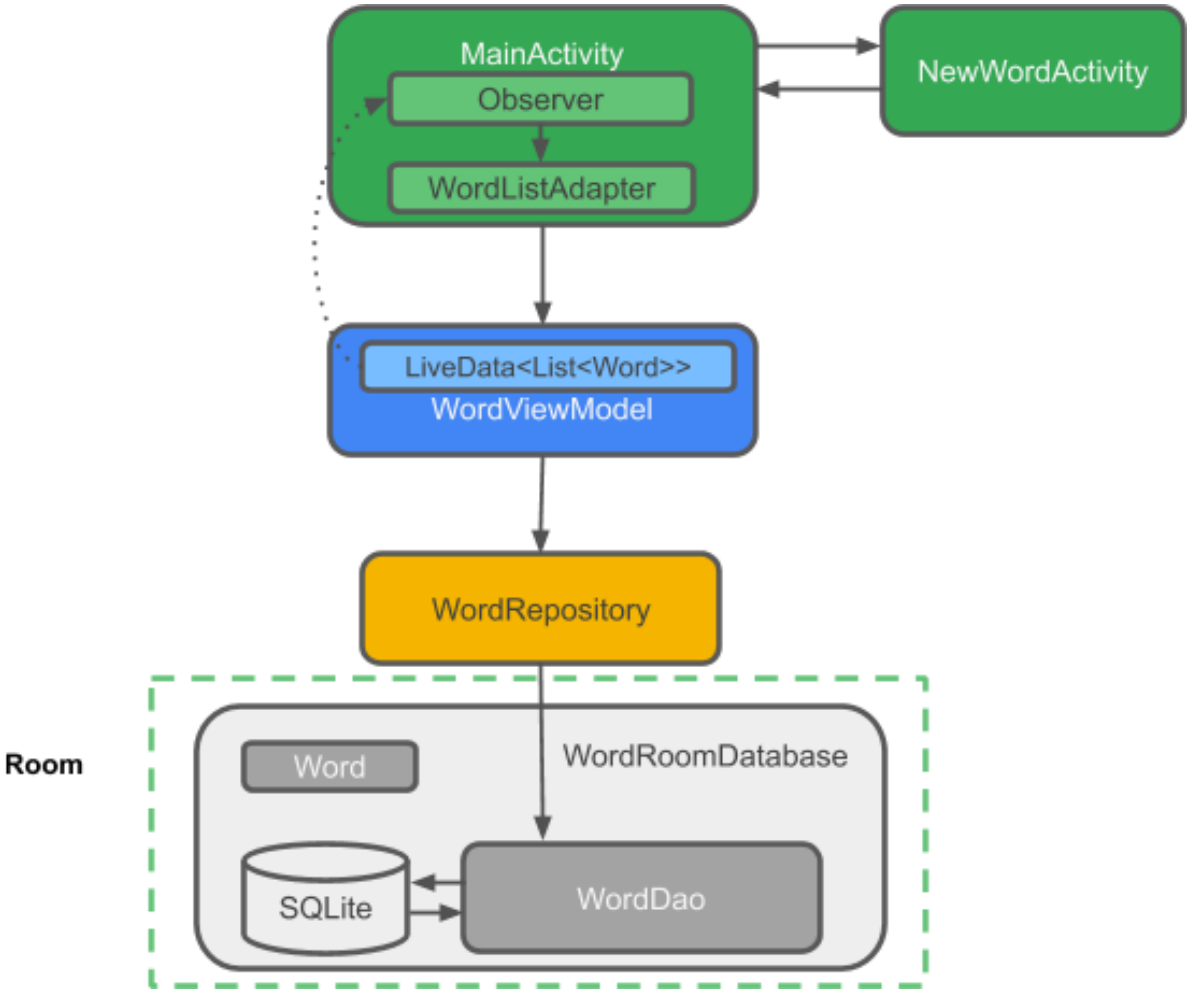


Рисунок 4.5 – Життєвий цикл даних

Репозиторії дозволяють здійснювати обмін інформацією між компонентами програми, віддаленою базою даних та локальною базою даних.

Активність для створення нового елементу може бути викликана з головного екрану, який містить список існуючих записів.

4.3 Спосіб зберігання даних у додатку

Важливість стійкого зберігання даних стає ще більш очевидною при врахуванні життєвого циклу типового додатка Android. З постійним ризиком того, що система виконання Android припинить компонент програми для звільнення ресурсів, всебічна стратегія зберігання даних, щоб уникнути втрати даних, є ключовим фактором при розробці та реалізації будь-якої стратегії розвитку додатків.

Тому при розробці програмного додатку було прийнято рішення на користь використання системи керування базами даних SQLite, що постачається в комплекті з операційною системою Android. Перш ніж заглиблюватися в специфіку SQLite, варто висвітлити короткий огляд баз даних та SQL.

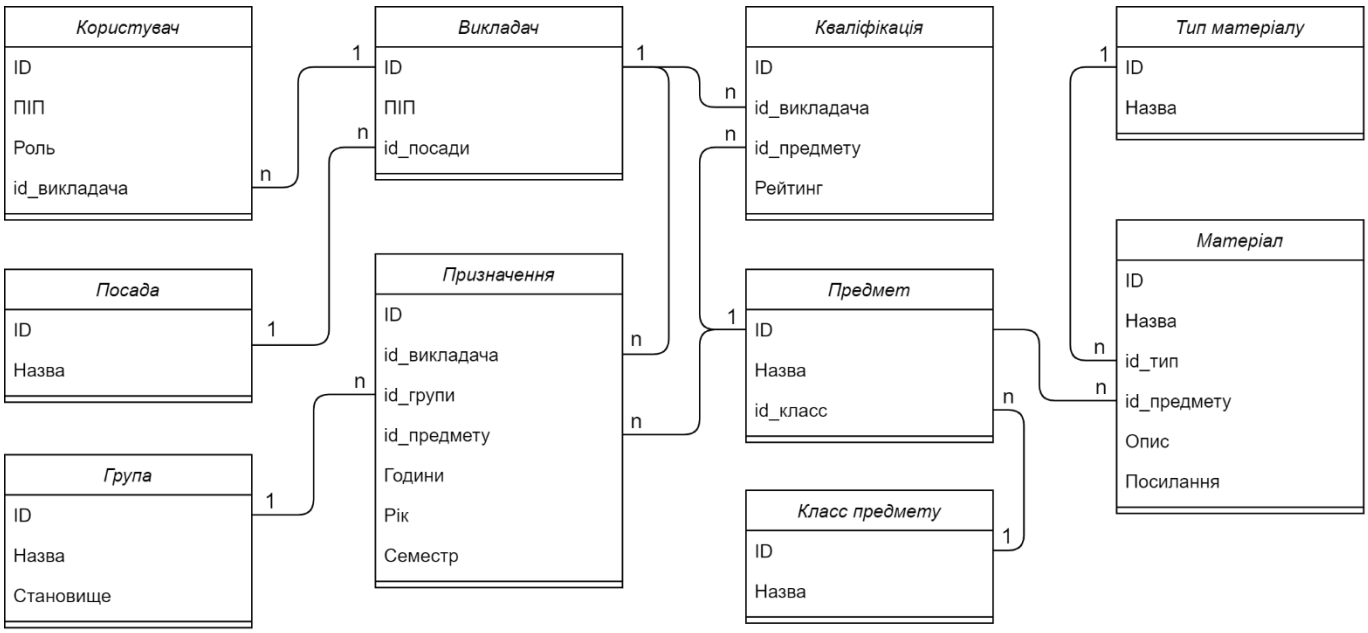


Рисунок 4.6 – Концептуальна модель бази даних

База даних складається з 10 таблиць (рисунок 4.6), що представляють дані, які можуть змінюватись у системі. Таблиці мають однакову структуру як в локальній так і у віддаленій базах даних. Ідентифікатор, характерний кожній таблиці представлений у вигляді унікального хеша та дає можливість однозначно ідентифікувати елемент.

Деякі з таблиць містять посилання на елементи з інших таблиць, завдяки унікальному ідентифікатору.

Наприклад елемент таблиці призначення містить інформацію про те, який викладач читає конкретний предмет для певної групи, а також містить додаткову важливу інформацію таку як кількість годин, рік та семестр викладання.

Схема баз даних визначає характеристики даних, що зберігаються в таблиці бази даних (рисунок 4.7). Схеми також використовуються для визначення структури цілих баз даних та взаємозв'язку між різними таблицями, що містяться в кожній базі даних.

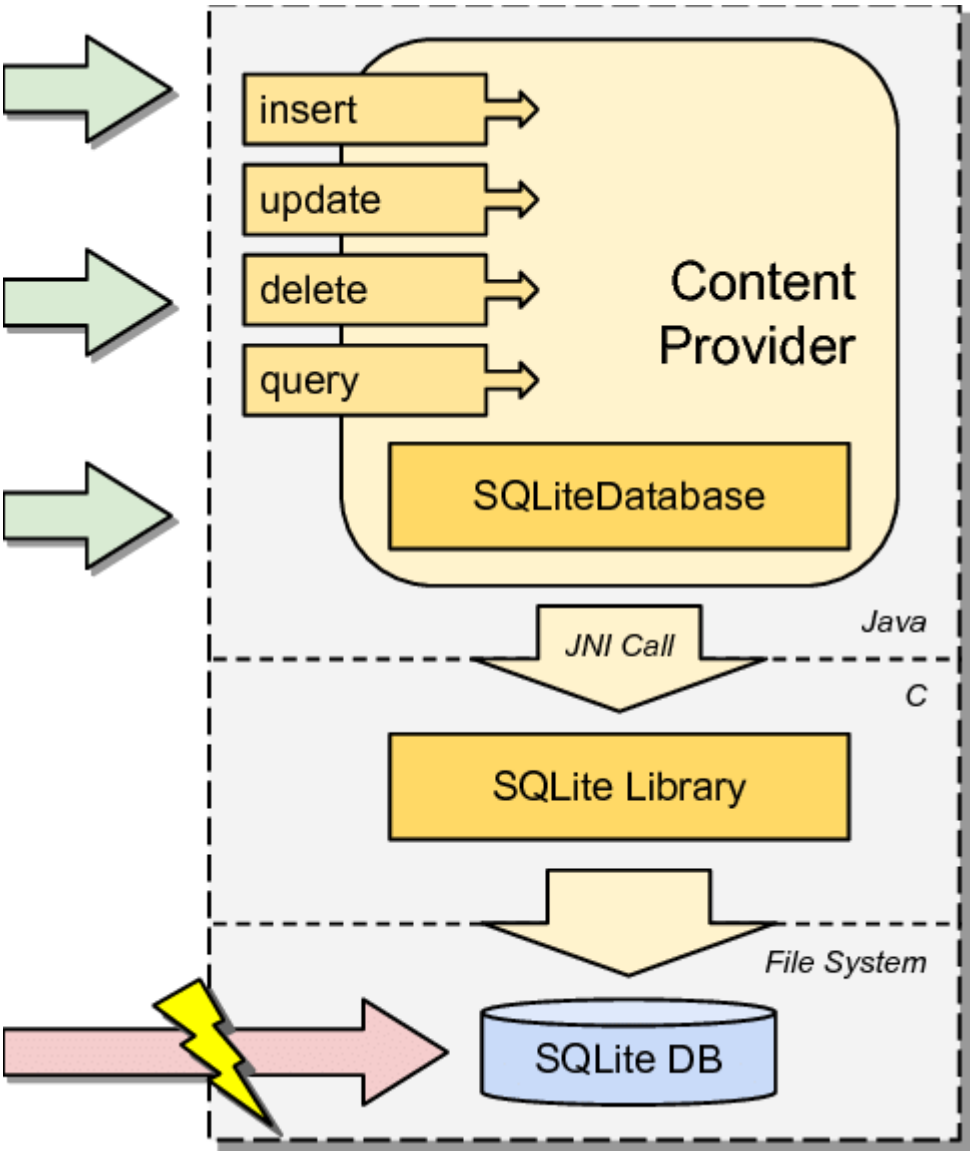


Рисунок 4.7 – Взаємодія та потік даних SQLite

SQLite - це вбудована реляційна система керування базами даних (RDBMS, з англ. Relational Database Management System). Більшість реляційних баз даних - це автономні серверні процеси, які працюють незалежно та у парі з програмами, які

потребують доступу до бази даних. SQLite є вбудованою базою, оскільки вона надається у вигляді бібліотеки, яка пов'язана з мобільними додатками. Таким чином, не існує окремого сервера баз даних, що працює у фоновому режимі.

SQLite написаний мовою програмування на C і, таким чином, Android SDK забезпечує «обгортку» на основі Java навколо базового інтерфейсу бази даних. Це по суті складається з набору класів, які можуть бути використані в коді Java програми для створення та керування базами даних на базі SQLite.

Обране рішення надає можливість для нормалізації та денормалізації програмних структур у об’єкти бази даних.

Доступ до даних здійснюється в базах даних SQLite за допомогою мови високого рівня, відомої як Структурована мова запитів (SQL). Це стандартна мова, яка використовується більшістю реляційних систем керування базами даних. SQLite в основному відповідає стандарту SQL-92.

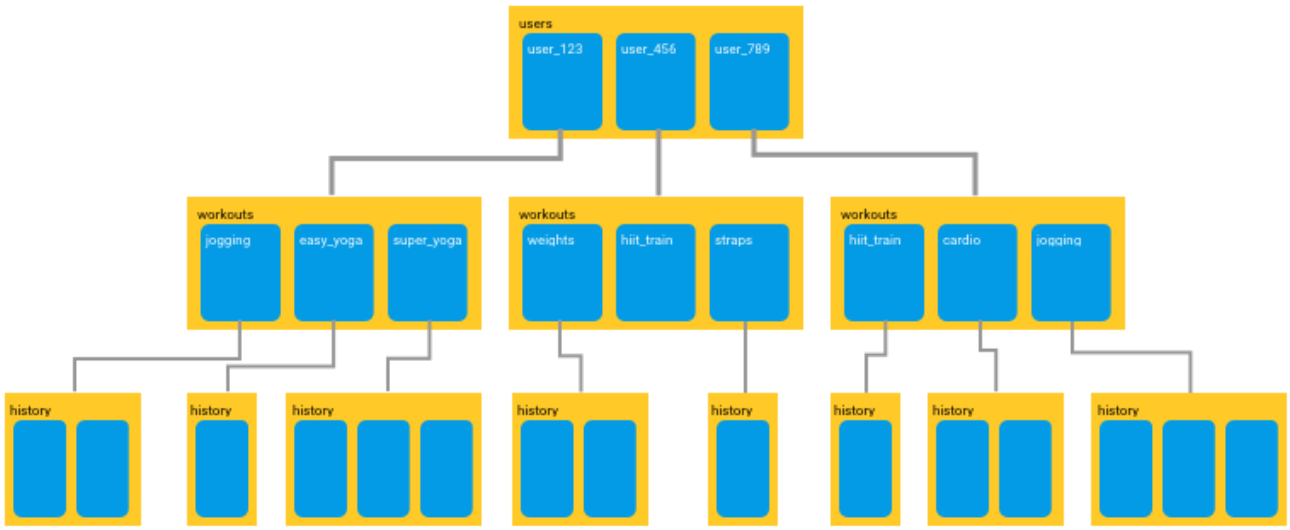
По своїй суті, мова SQL не є складною у використанні і розроблена спеціально для забезпечення можливості читання та запису даних у базу даних. Оскільки SQL містить невеликий набір ключових слів, його можна швидко вивчити. Крім того, синтаксис SQL є майже ідентичним серед більшості реалізацій СУБД.

4.4 Централізований доступ до даних

Для того щоб кожен користувач мобільного додатку міг отримувати дані з єдиного центру інформації потрібна була гнучка, масштабована база даних. Такою є хмарна база даних Cloud Firestore від Firebase. Вона підтримує синхронізацію даних між клієнтськими додатками через слухачів у реальному часі та пропонує автономну підтримку для мобільних пристроїв через Інтернет, даючи цим створювати чуйні додатки, які працюють незалежно від затримки мережі чи підключення до Інтернету.

Cloud Firestore - це база типу документ-модель (document-model). Це означає, що всі ваші дані зберігаються в об'єктах, які називаються документами, у вигляді ключ-значення. Де значення може бути чим завгодно, від строкової змінної до бінарного файлу. Такі документи групуються в колекції. [6]

Така база даних, може містити одну або декілька колекцій з документами, які будуть посилатися на інші «субколекції» зі своїми документами і так далі (рисунок



4.8).

Рисунок 4.8 – Зв’язки між колекціями та документами Cloud Firestore

Таким чином при взаємодії з інтерфейсом користувача, відправляються запити на отримання даних до моделі, яка в свою чергу робить запит на отримання цих даних за протоколом обміну даними та перенаправляє запит до репозіторія. Репозіторій відповідає за роботу з базами даних - як локальною, так і віддаленою (рисунок 4.9).

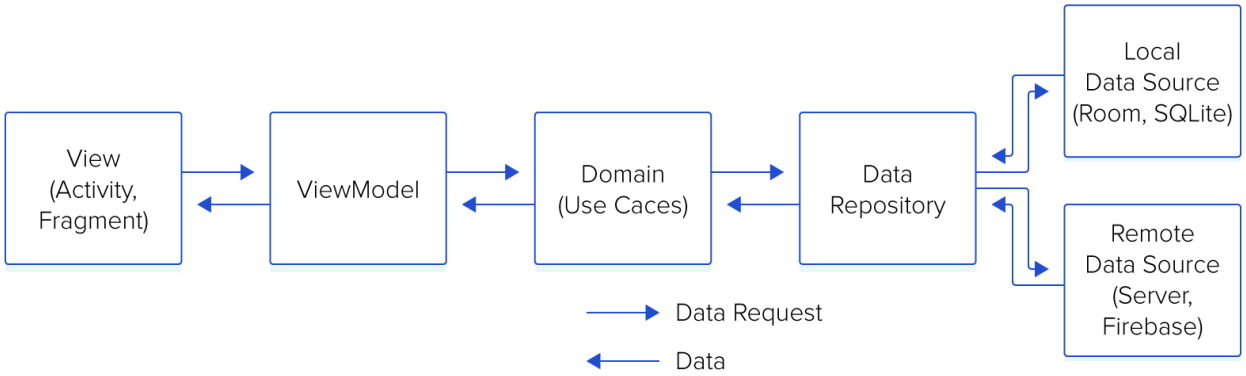


Рисунок 4.9 – Схема доставки даних компонентами системи

Як тільки запит був оброблений, відбувається передача даних, поки не досягне першого відправного пункту.

Нова структура даних дає декілька переваг при побудові запитів до бази.

По-перше, всі запити є неглибокими, що дає можливість просто витягти документ без необхідності тягти все дані, що містяться в будь-який з пов'язаних субколекцій. Це дає можливість зберігати дані програми ієрархічно таким чином, що не доведеться турбуватися про завантаження масивів непотрібних даних.

Так наприклад документ *user_456* може бути витягнутий без захоплення будь-яких документів у нижчестоящих субколекціях. [4]

Для прикладу, система запитів Cloud Firestore дозволить знайти 10 кращих ресторанів в Чикаго однаково швидко, будь у базі 300, 300 тисяч або 30 мільйонів ресторанів. На презентації один з інженерів сказав «У Cloud Firestore просто неможливо зробити малопродуктивний запит».

Завдяки хмарному сховищу, користувачі можуть отримувати інформацію з різних пристроїв, незалежно від їх місцезнаходження (рисунок 4.10).

Рисунок 4.10 – Схема комунікації Cloud Firestore

Це рішення також дає змогу автоматично синхронізувати дані між пристроями. Користувачі можуть отримати доступ до своїх даних та змінити їх у будь-який час, навіть коли вони в режимі офлайн. Це дає можливості для синхронізації даних.

4.3 Інтерфейс користувача

Інтерфейс користувача реалізований за допомогою засобів Android Studio 3.6.3. Головним структурним компонентом є діяльність (Activity), яка складається з компонентів Fragment або ViewGroup. Інтерфейс користувача визначається у файлі xml. Під час компіляції кожен елемент XML компілюється в еквівалентний клас Android GUI з атрибутами, представленими методами. Діяльність складається з відображень (View). Відображення - лише віджет, який з'являється на екрані. Це може бути кнопка, випадаючий список тощо. Одне або більше представлень можна об'єднати в один компонент GroupView, який базується на одному з макетів (Layout).

Редактор макета дозволяє швидко створювати макети, перетягуючи елементи інтерфейсу в редактор візуального дизайну, а не писати макет XML вручну. Редактор дизайну дає змогу попередньо переглянути макет на різних пристроях та версіях Android, а також динамічно змінити розмір макета, щоб переконатися, що він добре працює на різних розмірах екрана. [9]

Редактор макета з'являється, якщо відкрити файл макета XML (рисунок 4.11).

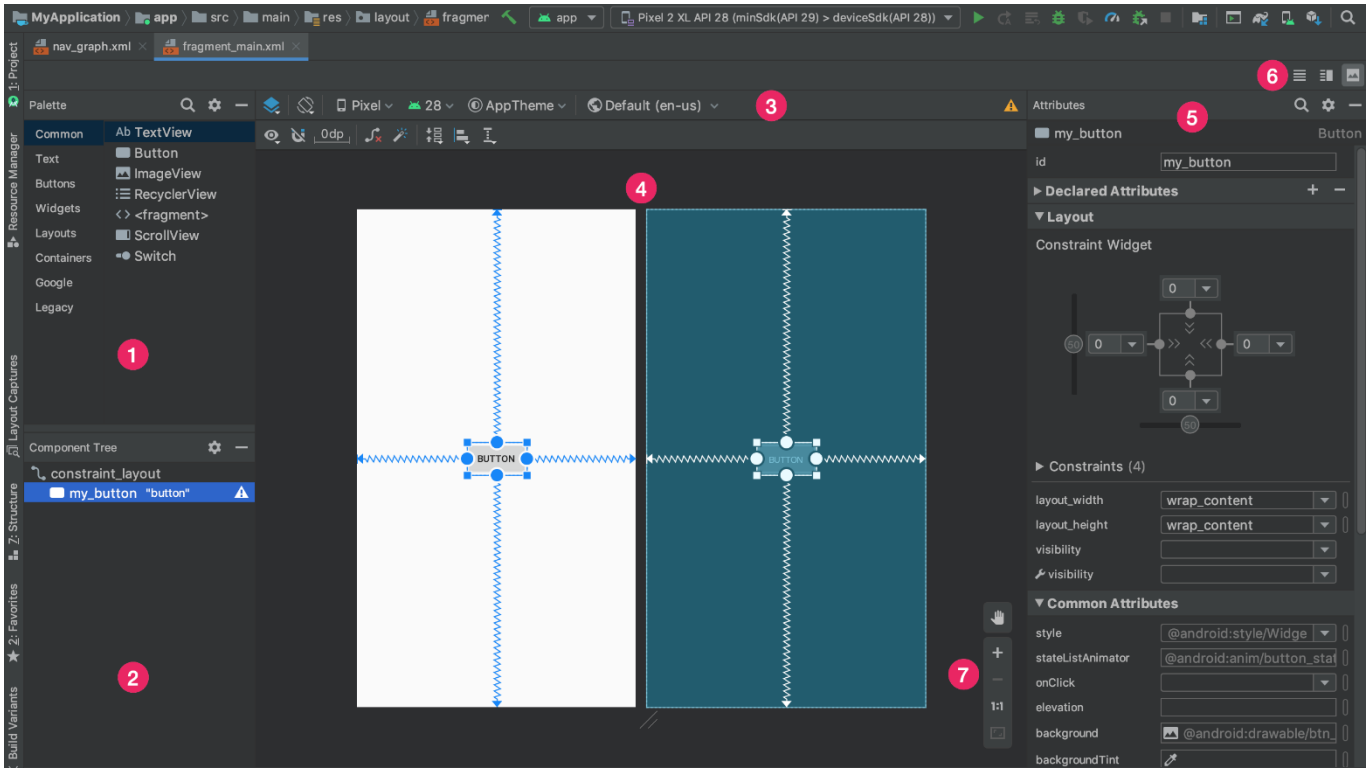


Рисунок 4.11 – Редактор макета

1. Палітра: містить різні представлення та групи представлень, які можна використати у своєму макеті.
2. Дерево компонентів: показує ієрархію компонентів у розроблюваному макеті.
3. Панель інструментів: дозволяє налаштувати зовнішній вигляд макета в редакторі та змінити атрибути макета.
4. Редактор дизайну: відредагуйте макет через представлення дизайну, відображення плану або в обох випадках.
5. Атрибути: надають можливість керувати атрибутами вибраного перегляду.
6. Режим перегляду: можливість перегляду макету в режимах написання коду, дизайну або розділеному. У розділеному режимі одночасно відображаються вікна написання коду та дизайну.
7. Елементи масштабування та панорамування: дозволяють регулювати розмір та положення попереднього перегляду в редакторі.

5. РОБОТА КОРИСТУВАЧА З ПРОГРАМОЮ

5.1 Запуск системи та системні вимоги

Для запуску розробленого додатку необхідно:

1. Версія операційної системи Android 7.0 (Nougat) або вище
2. Завантажити мобільний додаток.
3. Встановити його.
4. Переконатись у наявності підключення до мережі Інтернет.

5.2 Функції системи та користувачів

Користувачі можуть мати одну з 5 ролей: Гість, Викладач, Методист, Адміністратор та Власник системи. Так, що кожна наступна роль має усі можливості попередньої плюс деякі додаткові.

Діаграма прецедентів (рисунок 5.1) відображає дії, які доступні користувачам системи.

З діаграми видно, що Гість може лише переглядати 4 основних категорії інформації. В той час як Викладач може додатково переглядати категорії навчальних дисциплін та матеріалів, а також додавати нові посилання на навчальні матеріали.

Також користувач має можливість вручну відправити запит на отримання даних з головної бази, якщо має підозру, що поточна інформація не є актуальною, наприклад через помилку з підключенням до мережі.

Методисту, на відміну від попередніх ролей, доступний функціонал аналітики. Він дозволяє робити вибірку викладачів за кваліфікаційним рейтингом, що базується на основі таких критерій як наявність публікацій, патентів, наукових доповідей, тощо.

Користувач, що має роль Адміністратора може отримати доступ до всіх даних та має можливість керування цими даними. За винятком інформації про існуючих користувачів, такої як Повне ім’я та Роль.

Власник додатку має всі права Адміністратора, а також може змінювати та видавати ролі користувачам.

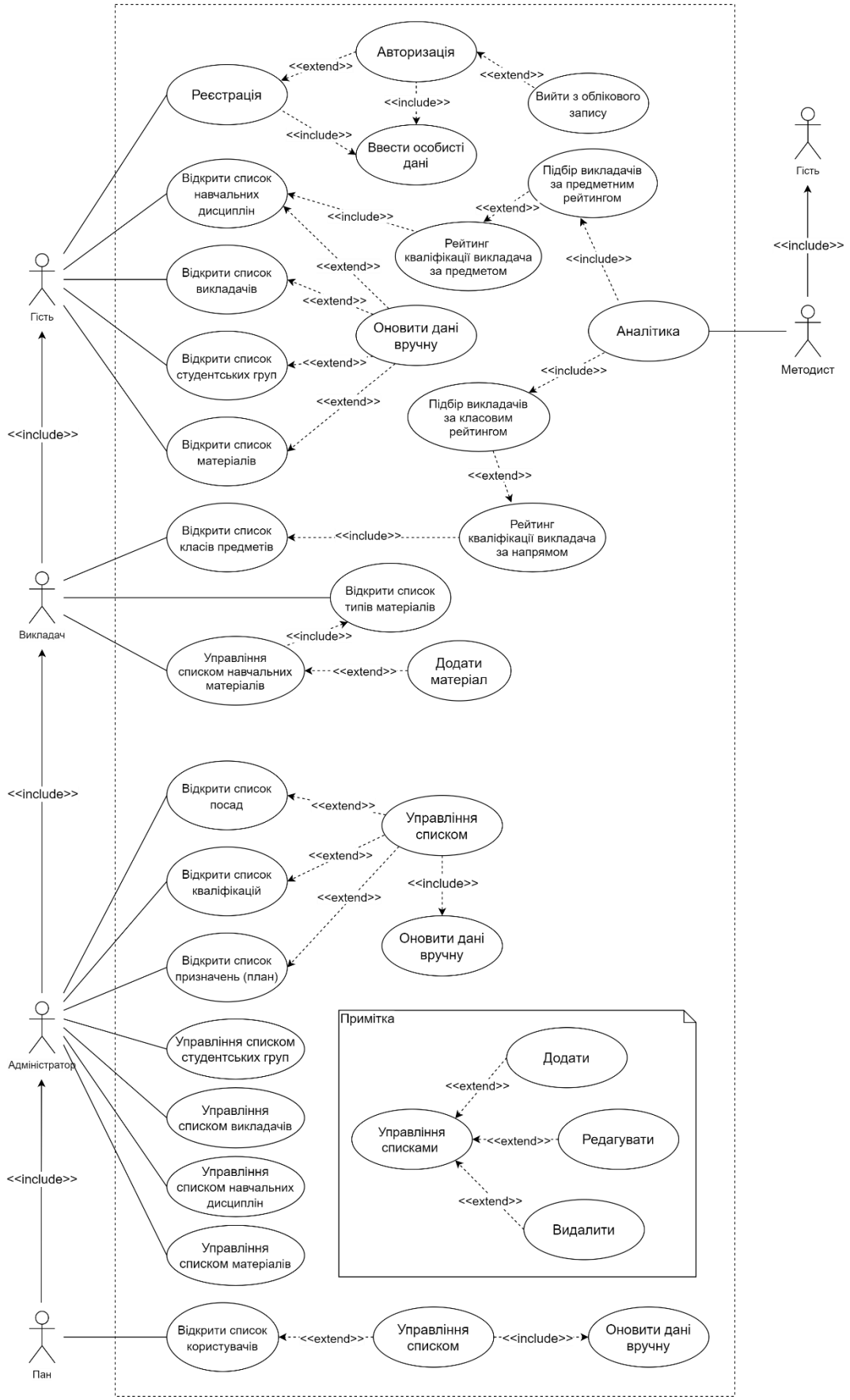


Рисунок 5.1 – Діаграма прецедентів

5.3 Сценарії роботи користувача з системою

При запуску додатку до уваги користувача представляється екран авторизації (рисунок 5.2).

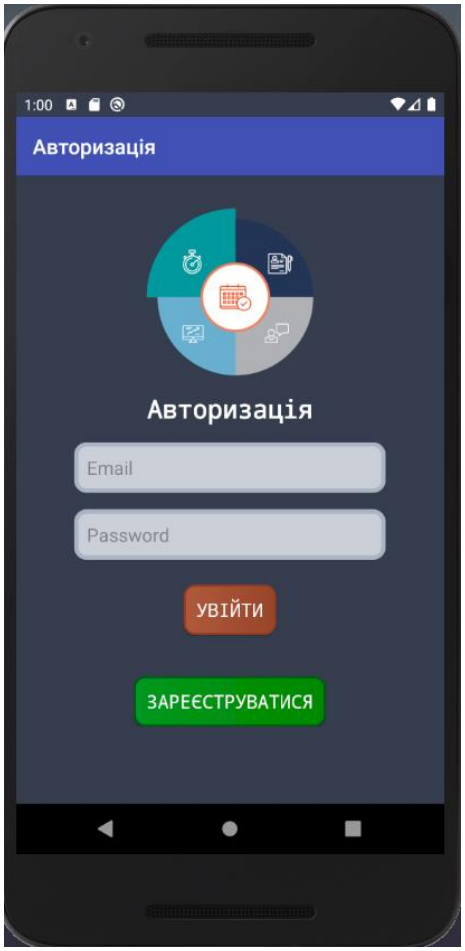


Рисунок 5.2 – Екран авторизації

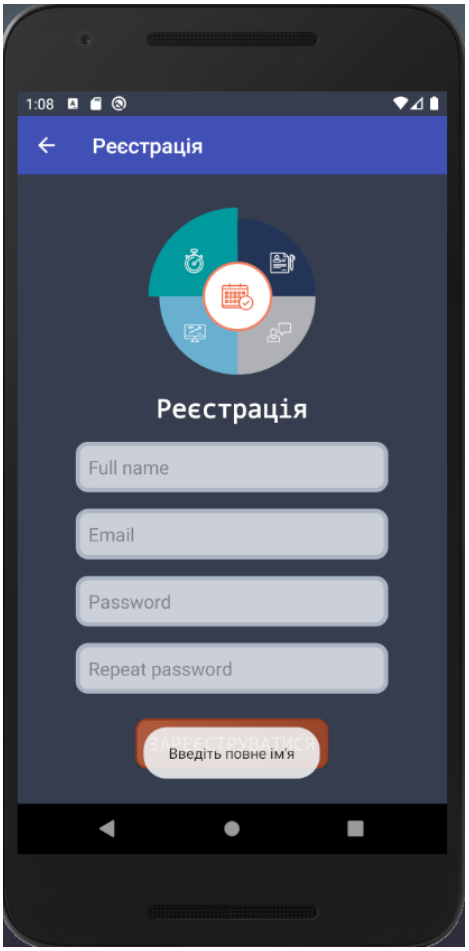


Рисунок 5.3 – Екран реєстрації

Якщо користувач вперше працює з цією системою, йому необхідно перейти на екран реєстрації для створення облікового запису в системі (рисунок 5.2).

Варто зазначити, що поля для авторизації та реєстрації мають свої вимоги. Наприклад, якщо користувач спробує зареєструватися, не вказавши при цьому своє повне ім’я, програма покаже повідомлення з підказкою у нижній частині екрану (рисунок 5.3).

Поле для вводу електронної адреси (Email) перевіряє правильність введеної адреси за всесвітнім офіційним стандартом RFC 5322 (рисунок 5.4). Пароль також має обмеження по мінімальній довжині, програма не дозволить ввести пароль, довжина якого складає менше ніж 6 символів (рисунок 5.5). Також, щоб переконатись що

користувач не зробив помилки при введені паролю, існує поле для його підтвердження. А у випадку якщо введені паролі не співпадають, користувач отримає попереджувальне повідомлення з інформацією про це (рисунок 5.6).

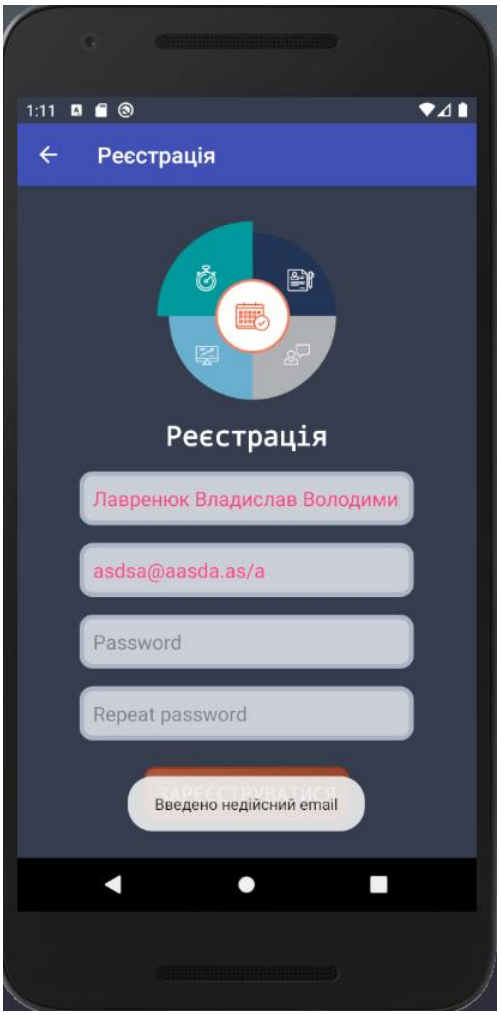


Рисунок 5.4 – Спроба реєстрації з недійсною електронною адресою

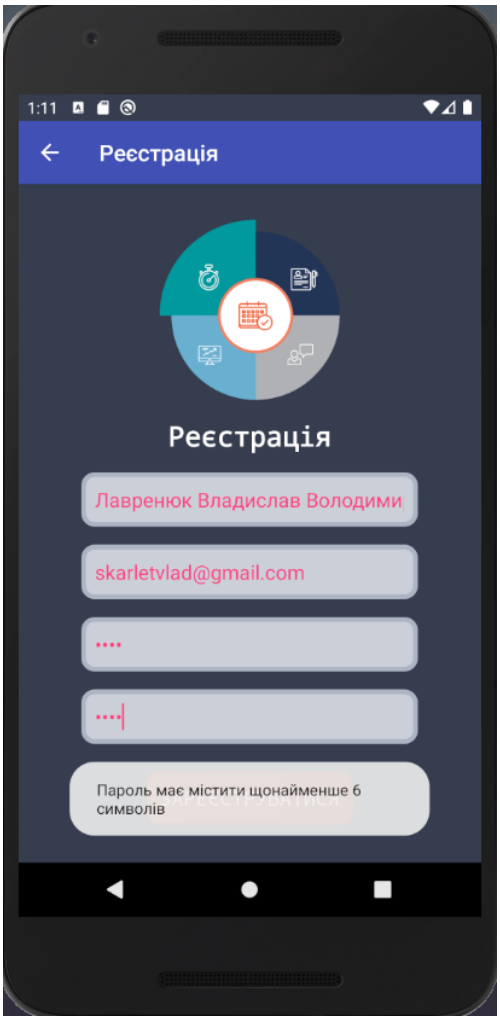


Рисунок 5.5 – Спроба реєстрації зі слабким паролем

Якщо користувач успішно зареєструвався, або авторизувався зі своїми обліковими даними, відбудеться перехід до головної сторінки кабінету користувача.

Варто зазначити, що якщо користувач був успішно авторизований, додаток це запам’ятає. Таким чином користувачу не потрібно вводити свої персональні дані, кожного разу коли він відкриває додаток, оскільки інформація про це буде зберігатись у локальному сховищі до тих пір, поки користувач не натисне кнопку виходу (рисунок 5.7), або почистить кеш додатку у налаштуваннях мобільного пристрою.

У персональному кабінеті, користувачу доступне меню навігації по екранам додатку, яке можна легко викликати жестом «свайп вправо». Важливим є те, що це меню може відрізнятись залежно від користувачів. Кожен користувач має свою роль, їх є 5: гість, викладач, методист, адміністратор та власник додатку.

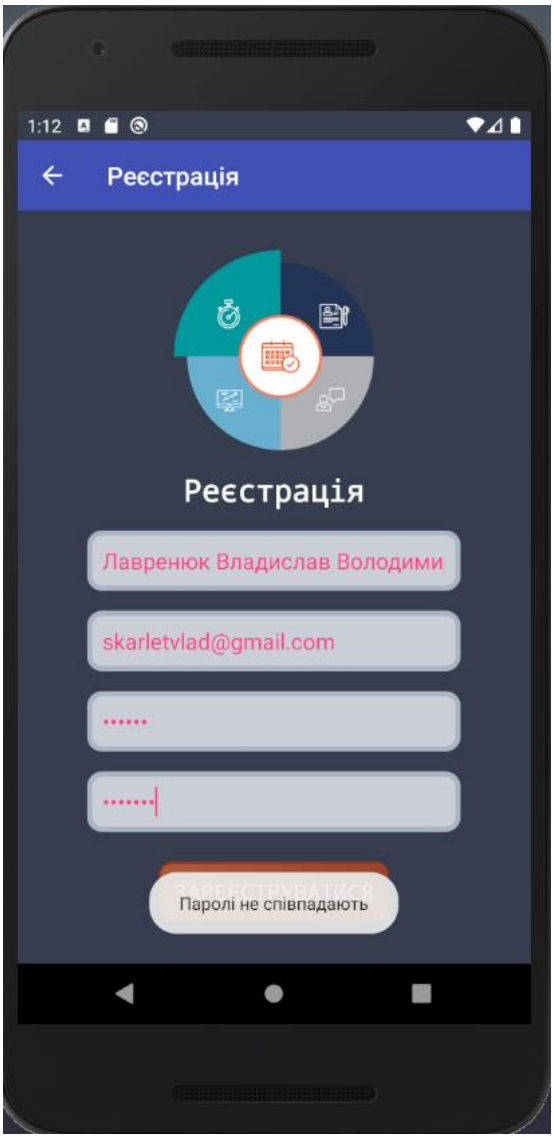


Рисунок 5.6 – Спроба реєстрації з помилкою підтвердження пароля.

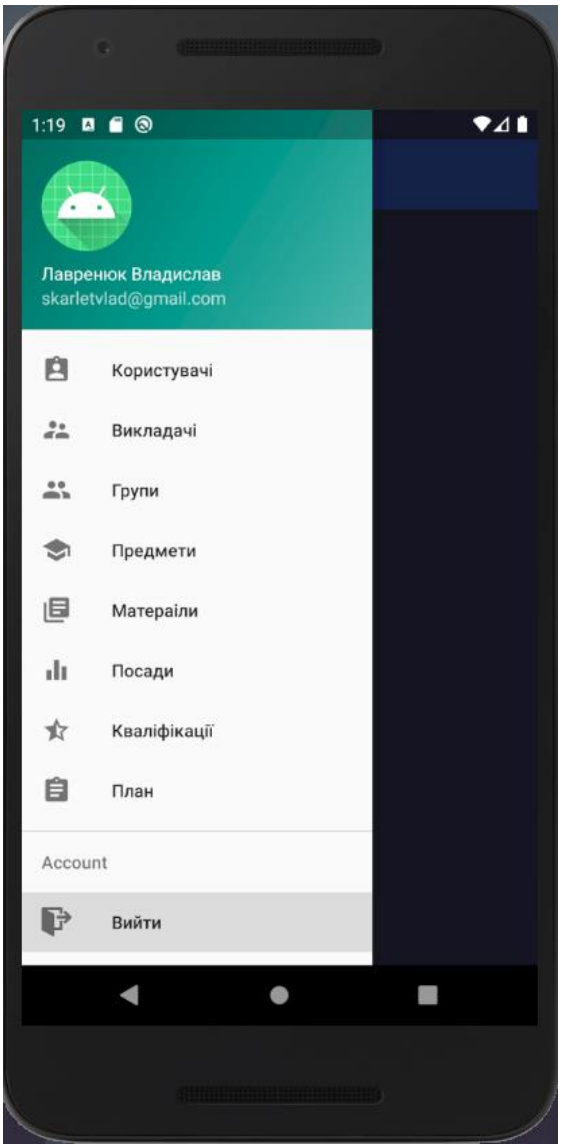


Рисунок 5.7 – Вихід з облікового запису через кабінет користувача

Власник має доступ до всього функціоналу, який може надати додаток. Призначенням цієї ролі є налагодження додатку у тих рідкісних випадках, коли навіть прав адміністратора виявиться недостатньо для цього. Під доступом мається на увазі права читання та запису будь-яких даних, що тільки існують у центральному сховищі даних.

Цю роль можна отримати лише через пряму взаємодію з центральним сховищем даних. Програмний додаток не дає можливості це зробити в цілях безпеки.

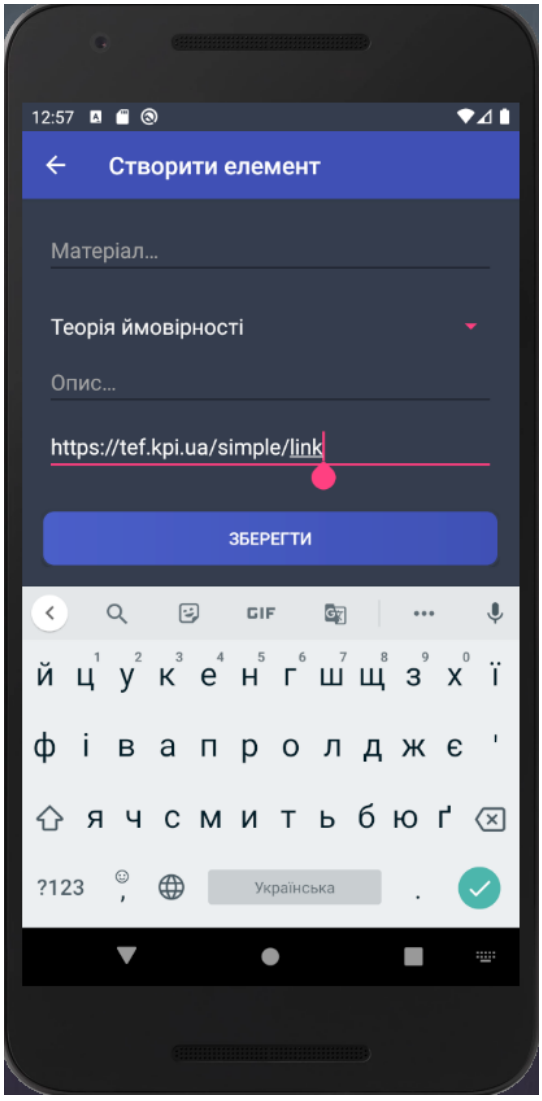


Рисунок 5.8 – Екран додавання нового матеріалу

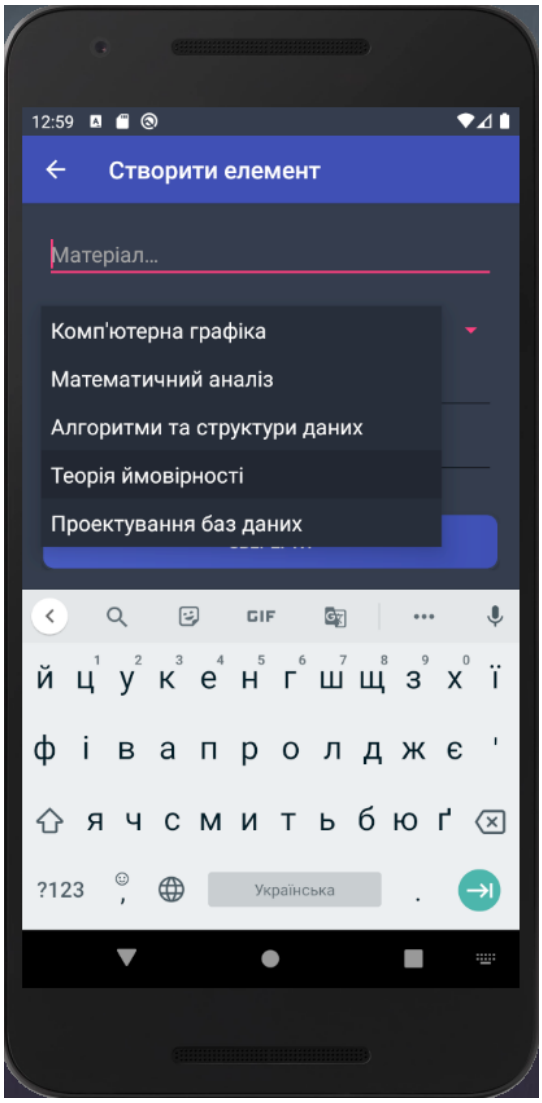


Рисунок 5.9 – Екран додавання нового матеріалу

Адміністратор має майже такий самий доступ до даних, за виключенням перегляду та редагування інформації про користувачів (примітка: пароль не зберігається у базі даних у цілях безпеки та конфіденційності користувачів, тому ніхто немає можливості його побачити чи змінити). Ця роль може бути видана власником додатку будь-якому користувачу.

Викладач може переглядати більшість основних даних у режимі читання за винятком кваліфікацій та навчального плану. Натомість доступна лише частина цієї інформації, а саме та яка відноситься безпосередньо до цього викладача. Також користувачі з цією роллю можуть додавати посилання на матеріали без яких-небудь обмежень (рисунки 5.8 та 5.9). Ця роль може бути отримана при реєстрації у випадку, якщо вказане ім’я користувача існує серед списку викладачів у базі даних, або якщо в базу був доданий новий викладач з таким самим ім’ям як у поточного користувача, у тих випадках, якщо користувач ще не має даної або вищої ролі. В усіх інших випадках роль користувача буде встановлена як Гість.

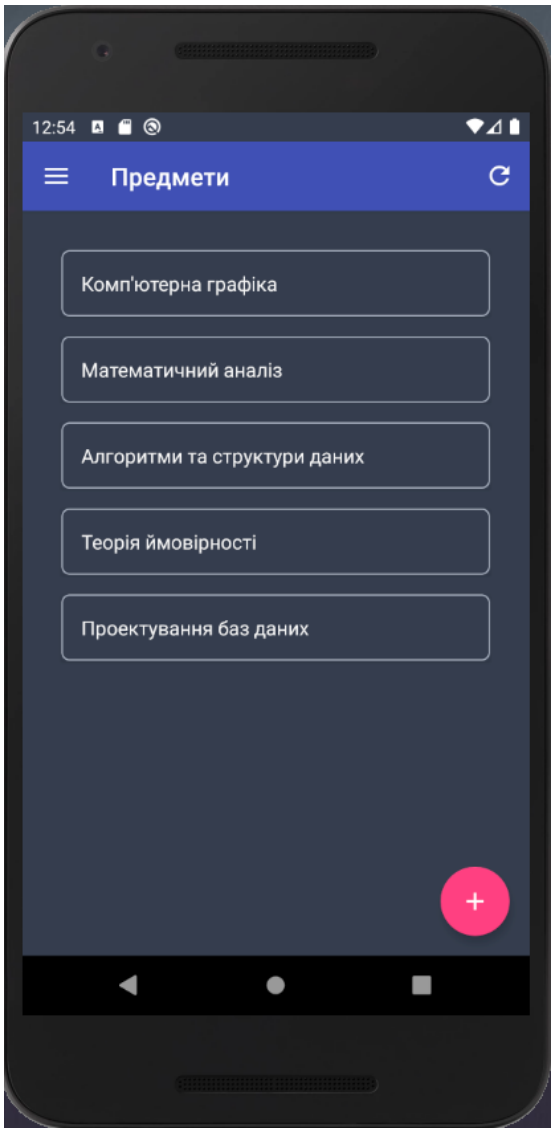


Рисунок 5.10 – Екран списку навчальних дисциплін

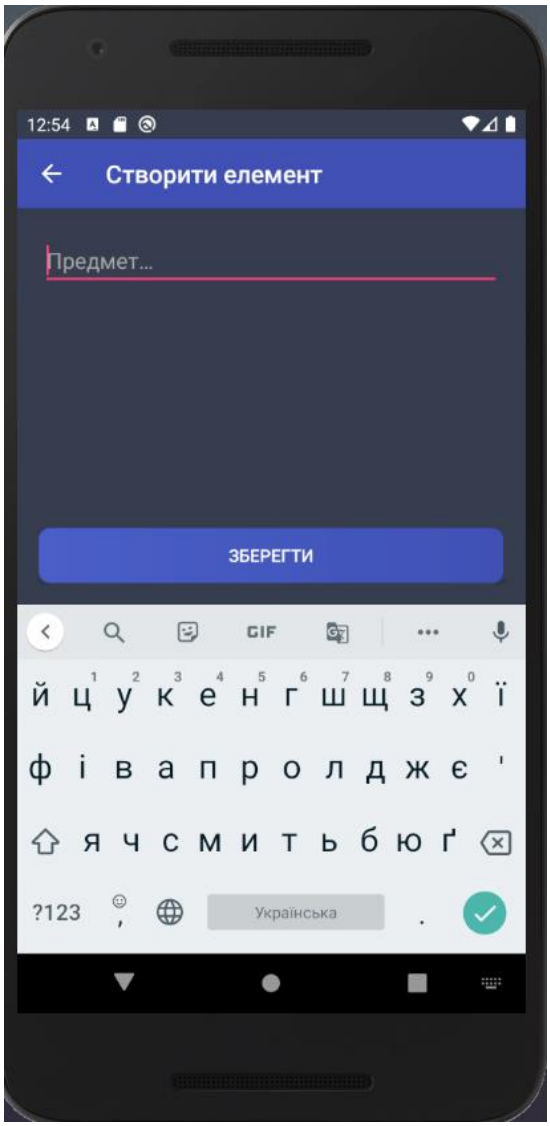


Рисунок 5.11 – Екран додавання нового предмету

Гість має найбільш обмежений доступ до даних. Користувач з такою роллю може лише переглядати дані, як список студентських груп, список викладачів та матеріали.

Скриншоти далі відображають функціонал, доступний власнику додатку, оскільки він має найбільші права доступу до системи. Що стосується керування даними, додаток надає зручний інтерфейс для таких цілей.

Наприклад якщо користувач зараз знаходиться на вікні перегляду навчальних дисциплін (рисунок 5.10). В правому нижньому кутку ми можемо бачити кнопку для додавання нового елементу до списку і, звичайно, до бази даних також. Натиснувши на неї ми потрапимо до вікна створення нового елементу (рисунок 5.11). Так як навчальна дисципліна не має інших власних характеристик окрім своєї назви (зв’язки є складовими інших об’єктів), то є лише поле для її вводу.

Окрім кнопки для додавання нового елементу на панелі верхнього меню є кнопки для відкриття меню користувача (теж саме що й свайп вправо) та кнопка для запиту на оновлення списку елементів, якщо по якійсь причині користувач вважає, що бачить не актуальні дані, хоча всі списки оновлюються в режимі реального часу.

Також окрім можливості додати новий елемент, користувач може редагувати або видаляти вже існуючі. Звичайно, за наявності ролі яка дозволяє це зробити, так само як і у випадку додавання елементів. В іншому випадку ці кнопки просто будуть недоступні і навіть не будуть видимі користувачу.

Щоб відредагувати (видалити) елемент списку, потрібно відтягнути потрібний елемент вправо (вліво), як показано на рисунках 5.12 та 5.13.

При видаленні елемент зникне зі списку та локального сховища даних, а також буде відправлено запит на видалення цього об’єкту з головної бази даних.

При натисканні на кнопку редагування, користувач потрапляє на екран редагування елемента, який виглядає майже так само як і екран створення, за винятком заповнених відповідних полів, заголовка екрану та дії яка відбудеться при натисканні на кнопку «Зберегти» (рисунок 5.14).

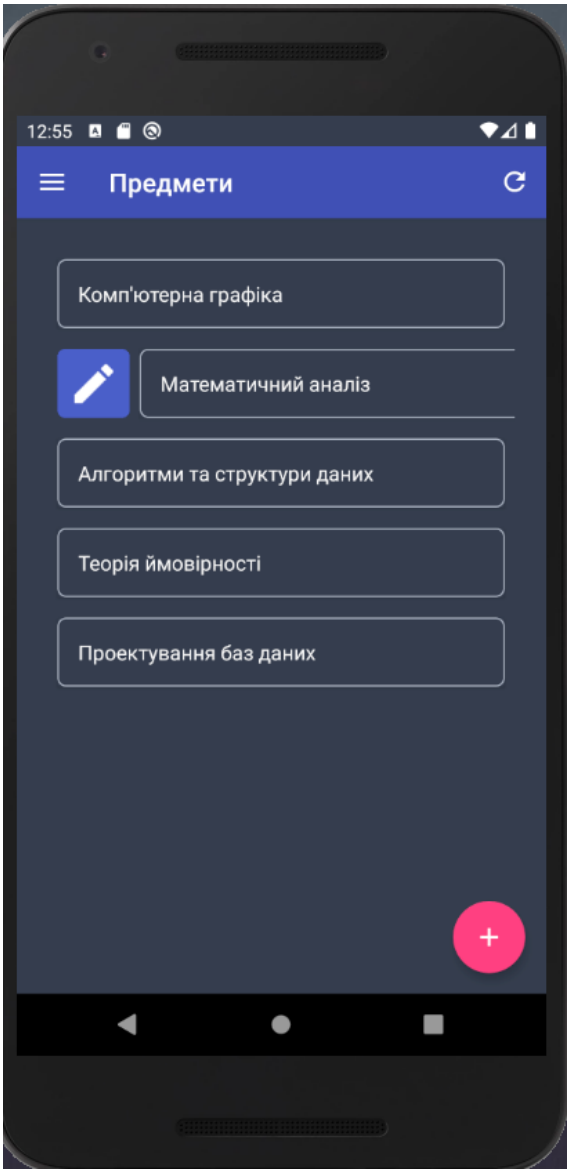


Рисунок 5.12 – Операція редагування навчальної дисципліни

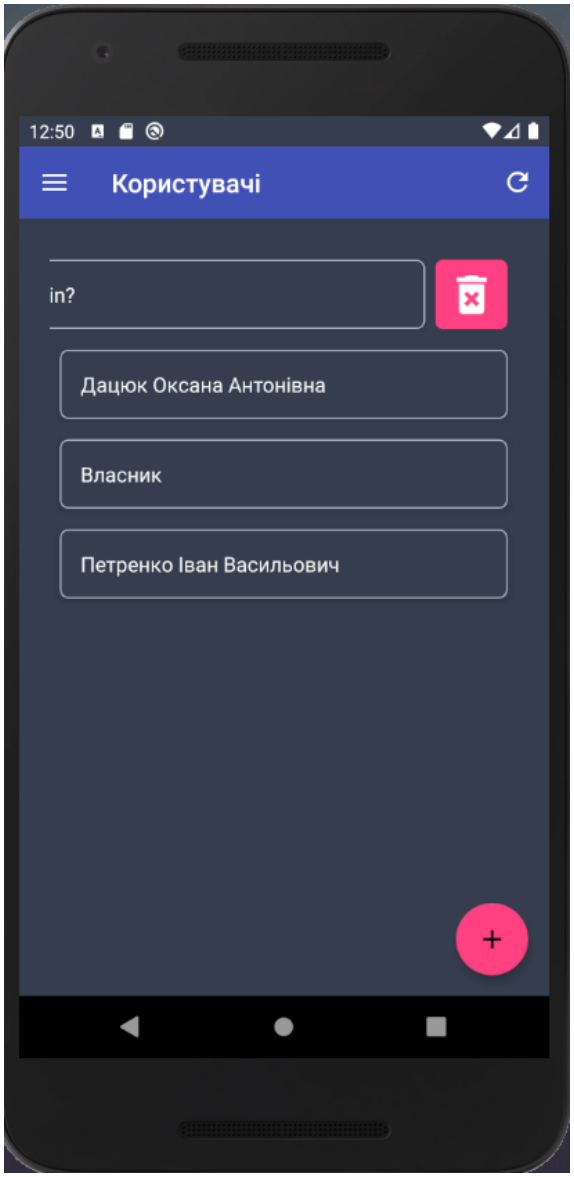


Рисунок 5.13 – Операція видалення користувача

Також більшість категорій елементів мають додаткову інформацію, яку можна переглянути натиснувши на нього (рисунок 5.15).

Рисунок 5.14 – Екран редагування навчальної дисципліни

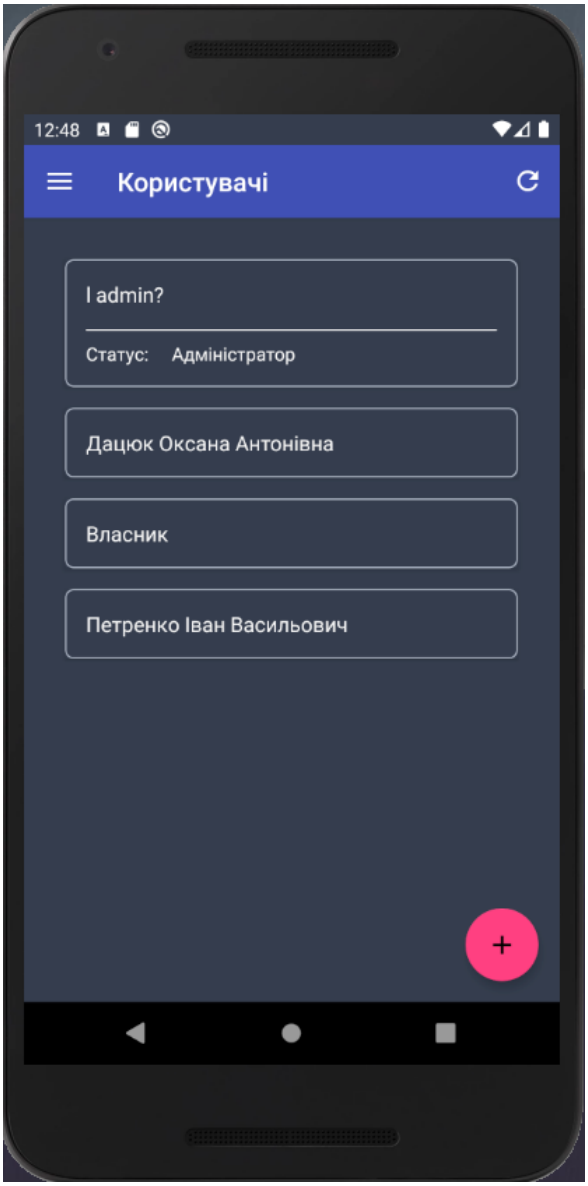


Рисунок 5.15 – Екран списку користувачів з розгорнутою інформацією про елемент

Усе це дає користувачу можливість отримати швидкий доступ до потрібної інформації через зручний, інтуїтивно зрозумілий інтерфейс мобільного додатку і виконує функції які лежать в основі інформаційної системи.

Окрім роботи з даними, система надає можливість проводити аналітику по вибору кваліфікаційних груп викладачів. Аналіз проводиться на основі рейтингу кваліфікацій. Рейтинг кваліфікацій визначається з урахуванням

критерій до ліцензійних вимог викладачів, зображених на рисунку 4.1, таких як патенти, стаж роботи, міжнародні експертизи, практичний досвід, наукові публікації, конференції, тощо.

В додатку, методист може отримати список кваліфікованих кадрів за вибраною навчальною дисципліною (рисунок 5.16), а також отримати аналітичні дані за типом навчального напряму (рисунок 5.17).

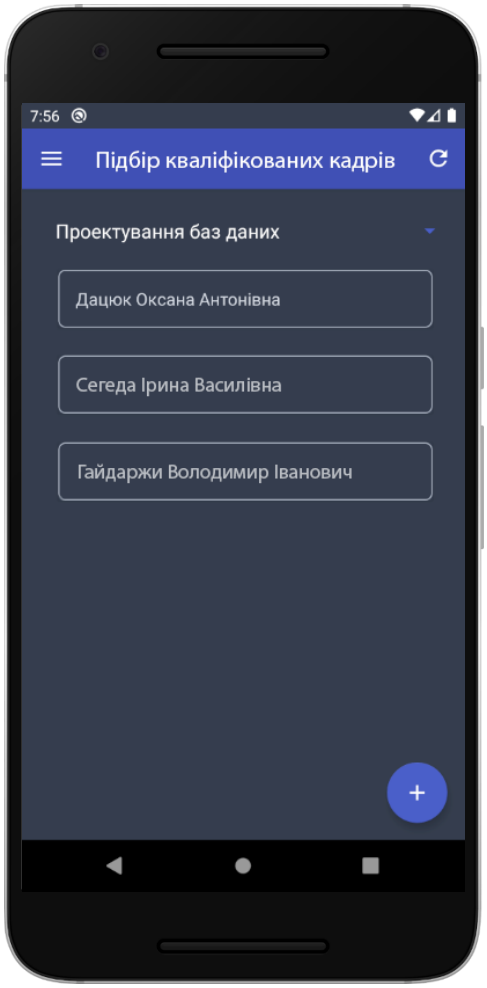


Рисунок 5.16 – Відображення сторінки підбору кваліфікованих кадрів

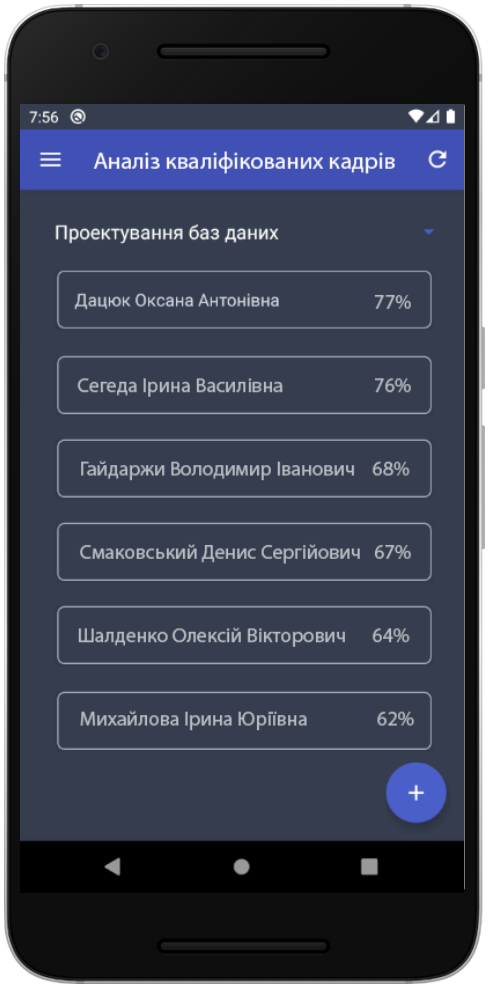


Рисунок 5.17 – Екран аналізу кваліфікованих кадрів за класом навчальної дисципліни

Відсоток напроти імен викладачів показує рейтинг кваліфікованості конкретної особи. Рейтинг складається відповідно до наявності критерій ліцензійних вимог, а також кількості кожної з них за спеціальною формулою. Варто зазначити, що головним фактором є наявність різних кваліфікаційних груп, а тільки потім – їх кількість.

ВИСНОВКИ

При розробці програмного продукту було переглянуто існуючі рішення для взаємодії з хмарними технологіями та керування контентом мобільних додатків. Проаналізовано їх переваги і недоліки.

В рамках виконання роботи були проаналізовані вимоги до кваліфікації викладачів на базі ліцензійних вимог професійної діяльності особи за спеціальністю та спроектована інформаційна модель бази даних.

Розроблено проект мобільної інформаційної системи вибору кваліфікаційних груп забезпечення викладачів. Була створена база даних за допомогою засобів Firebase з використанням хмарних технологій.

Реалізовано програмне забезпечення, яке дозволяє виконувати підбір інформаційних списків викладачів для вказаного предмету, та проводити аналіз вибору кваліфікаційних груп відповідно до ліцензійних вимог.

Побудовано систему, яка використовує декілька джерел даних для своєї роботи, та відповідає вимогам побудови інтерфейсів та створення мобільних додатків з використанням сучасних технологій.

Результатом роботи є розроблене програмне рішення та забезпечує актуальність інформації у реальному часі.

Створений продукт являється мобільним додатком, а тому потребує доступ до ресурсів Інтернет, для підтримання актуальності даних та можливості вносити зміни до єдиного інформаційного центру системи.

Програма дає можливість швидко отримувати потрібну інформацію незалежно від місця знаходження користувача та наявності підключення до мережі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Room Persistence Library [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/topic/libraries/architecture/room.
2. LiveData [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/topic/libraries/architecture/livedata.
3. Appropriate Uses For SQLite [Електронний ресурс] – Режим доступу до ресурсу: https://www.sqlite.org/whentouse.html.
4. Get started with Cloud Firestore – Режим доступу до ресурсу: https://firebase.google.com/docs/firestore/quickstart.
5. Android KTX [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/kotlin/ktx.
6. Database Cloud Firestore [Електронний ресурс] – Режим доступу до ресурсу: https://console.firebase.google.com/u/0/project/_/firestore/users.
7. Android View Model [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/topic/libraries/architecture/viewmodel.
8. Handling Lifecycles with Lifecycle-Aware Components [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/topic/libraries/architecture/lifecycle.
9. Android Studio release notes [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/studio/releases.

ДОДАТОК 1

Аналітична система вибору кваліфікаційних груп забезпечення дисциплін

СПЕЦИФІКАЦІЯ

УКР.КПІім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б

Аркушів 2

Київ 2020

Позначення	Найменування	Примітки
Документація		
УКР.КПім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б 81-1	121_БДР_ЛавренюкВВ_2020_Записка.docx	Текстова частина дипломної роботи
Компоненти		
УКР.КПім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б 12-1	StaffScheduleSystem.apk	Основний файл додатку аналітичної системи
УКР.КПім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б 12-2	DataModule.aar	Модуль програми, що відповідає за передачу даних в системі
УКР.КПім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б 12-3	NetworkModule.aar	Модуль програми, що відповідає за передачу даних через Інтернет
УКР.КПім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б 12-4	CoreModule.aar	Модуль програми, що інкапсулює основну бізнес-логіку системи
УКР.КПім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б 12-5	UseCasesModule.aar	Модуль програми, що оперує взаємодією компонентів системи

ДОДАТОК 2

Аналітична система вибору кваліфікаційних груп забезпечення дисциплін

ЛІСТИНГ ПРОГРАМНОГО МОДУЛЯ

УКР.КПІім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б_12-2

Аркушів 10

Київ 2020

```
package ua.kpi.skarlet.sss.data.repository.template

import android.util.Log
import com.google.firebase.firestore.FirebaseFirestore
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.launch
import ua.kpi.skarlet.sss.data.db.dao.AbstractDAO
import ua.kpi.skarlet.sss.data.db.dto.DataTransferObject
import ua.kpi.skarlet.sss.data.db.dto.MediateItem
import ua.kpi.skarlet.sss.data.db.entity.ItemEntity
import ua.kpi.skarlet.sss.data.local.constant.DOCUMENTS_RETRIEVED
import ua.kpi.skarlet.sss.data.local.constant.ERROR_GETTING_DOCUMENTS
import ua.kpi.skarlet.sss.data.local.constant.ID_DEFAULT
import ua.kpi.skarlet.sss.data.local.constant.REFRESH_ITEMS_CALLED
import java.lang.RuntimeException

abstract class AbstractRepository<Item>(protected val scope: CoroutineScope) {
    private val mTAG: String = this::class.simpleName!!
    abstract val collection: String

    protected val firestore = FirebaseFirestore.getInstance()

    protected abstract val mBridge: FirestoreBridge<out ItemEntity, out MediateItem>


    abstract suspend fun insert(item: Item)

    abstract suspend fun update(item: Item)

    abstract suspend fun delete(item: Item)

    protected fun checkId(id: String) {
```

```
        if (ID_DEFAULT == id) throw RuntimeException("Cannot make the operation without element ID")
    }

    /**
     * Refresh data from the repository. Use a coroutine launch to run in a
     * background thread.
     */
    fun refresh() {
        scope.launch(Dispatchers.IO) {
            refreshItems()
        }
    }

    fun refreshAndExecuteOnMain(action: () -> Unit) {
        scope.launch(Dispatchers.IO) {
            refreshItems(action)
        }
    }

    private fun refreshItems(action: (() -> Unit)? = null) = mBridge.refreshItems(action)

    protected inner class FirestoreBridge<Entity : ItemEntity, Mediate : MediateItem>(
        private val roomDao: AbstractDAO<Entity>,
        private val dto: DataTransferObject<Mediate, Entity>
    ) {

        /**
         * Refresh the users stored in the offline cache.
         *
         * This function uses the IO dispatcher to ensure the database insert database operation
         * happens on the IO dispatcher. By switching to the IO dispatcher using `withContext` this
         * function is now safe to call from any thread including the Main thread.
         */
```

```
internal fun refreshItems(action: (() -> Unit)?) {
    Log.d(mTAG, REFRESH_ITEMS_CALLED)
    val callback = { items: List<Mediate> ->
        scope.launch(Dispatchers.IO) {
            updateTable(items)
            scope.launch(Dispatchers.Main) {
                action?.invoke()
            }
        }
    }
    getAllFirestoreItems(callback)
}

private fun getAllFirestoreItems(
    callback: (List<Mediate>) -> Job
) {
    firestore.collection(collection)
        .get()
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                Log.i(mTAG, DOCUMENTS_RETRIEVED)

                @Suppress("UNCHECKED_CAST")
                val mediateModel: List<Mediate> =
                    (task.result?.let { mBridge.dto.asMediateModel(it) }
                        ?: emptyList()) as List<Mediate>

                callback(mediateModel)
            } else {
                Log.w(mTAG, ERROR_GETTING_DOCUMENTS, task.exception)
            }
        }
}
```

```
        @Synchronized
        private suspend fun updateTable(items: List<Mediate>) {
            Log.i(mTAG, "Items in the local database were updated")
            roomDao.resetInTransaction(dto.asRoomModel(items))
        }
    }
}
```

```
package ua.kpi.skarlet.sss.data.repository.template

import kotlinx.coroutines.*
import ua.kpi.skarlet.sss.data.db.dao.AbstractDAO
import ua.kpi.skarlet.sss.data.db.dto.DataTransferObject
import ua.kpi.skarlet.sss.data.db.dto.MediateItem
import ua.kpi.skarlet.sss.data.db.entity.ItemEntity

// Declares the DAO as a private property in the constructor. Pass in the DAO
// instead of the whole database, because only the access to the DAO is needed
abstract class ItemRepository<Mediate : MediateItem, Entity : ItemEntity>(
    scope: CoroutineScope,
    private val roomDao: AbstractDAO<Entity>
) : AbstractRepository<Entity>(scope) {

    // Room executes all queries on a separate thread.
    // Observed LiveData will notify the observer when the data has changed.
    protected abstract val dto: DataTransferObject<Mediate, Entity>
    override val mBridge by lazy { FirestoreBridge(roomDao, dto) }

    override suspend fun insert(item: Entity) {
        val mediateItem = dto.asMediateItem(item)

firestore.collection(collection).add(dto.asFirestoreItem(mediateItem)).addOnCompleteListener {
            scope.launch(Dispatchers.IO) {
                refresh()
            }
```

```
        }
    }

    override suspend fun update(item: Entity) {
        val mediateItem = dto.asMediateItem(item)
        checkId(mediateItem.id)
        firestore.collection(collection).document(mediateItem.id)
            .set(dto.asFirestoreItem(mediateItem))
        roomDao.update(item)
    }

    override suspend fun delete(item: Entity) {
        val mediateItem = dto.asMediateItem(item)
        checkId(mediateItem.id)
        firestore.collection(collection).document(mediateItem.id).delete()
        roomDao.delete(item)
    }
}
```

```
package ua.kpi.skarlet.sss.data.repository.template

import android.util.Log
import androidx.lifecycle.LifecycleOwner
import androidx.lifecycle.LiveData
import androidx.lifecycle.Observer
import kotlinx.coroutines.*
import ua.kpi.skarlet.sss.data.db.StaffRoomDatabase
import ua.kpi.skarlet.sss.data.db.dao.AbstractDAO
import ua.kpi.skarlet.sss.data.db.dto.DataTransferComplexObject
import ua.kpi.skarlet.sss.data.db.dto.MediateItem
import ua.kpi.skarlet.sss.data.db.dto.MediateProxy
import ua.kpi.skarlet.sss.data.db.entity.ItemEntity

class LiveProperty<Entity : ItemEntity>(
    private val dao: AbstractDAO<Entity>
```

```
) {
    @Volatile
    private var ids: List<String>
    private var items: LiveData<List<Entity>>
    val all: ArrayList<Entity>

    init {
        ids = emptyList()
        all = ArrayList(ids.size)
        items = dao.getItemsByIDs(ids)
    }

    @Synchronized
    private fun refreshAll(items: List<Entity>) {
        all.clear()
        all.addAll(
            sortInIdsOrder(
                ids,
                items
            )
        )
    }

    @Synchronized
    fun changeIds(ids: List<String>, lifecycleOwner: LifecycleOwner, callback: () -> Unit = {}) {
        if (this.ids.size != ids.size || !this.ids.containsAll(ids)) {
            this.ids = ids
            if (items.hasObservers()) {
                items.removeObservers(lifecycleOwner)
            }
            items = dao.getItemsByIDs(this.ids)
            items.observe(lifecycleOwner, observer(callback))
        }
    }
```

```
    @Synchronized
    fun observer(callback: () -> Unit) = Observer<List<Entity>?> { items ->
        items?.let {
            refreshAll(it)
            callback()
        }
    }

    companion object {

        private fun <Entity : ItemEntity> sortInIdsOrder(
            idsI: List<String>,
            allI: List<Entity>
        ): List<Entity> {
            val sorted = ArrayList<Entity>(idsI.size)
            idsI.forEach { id ->
                allI.find { id == it.id }?.let { item ->
                    sorted.add(item)
                }
            }
            return sorted
        }

    }
}

abstract class CombinedRepository<Entity : ItemEntity, Mediate : MediateItem, Proxy: MediateProxy>(
    scope: CoroutineScope,
    private val roomDao: AbstractDAO<Entity>
) : AbstractRepository<Proxy>(scope) {

    // Room executes all queries on a separate thread.
```

```
// Observed LiveData will notify the observer when the data has changed.
protected abstract val dto: DataTransferComplexObject<Proxy, Mediate, Entity>
override val mBridge by lazy { FirestoreBridge(roomDao, dto) }

protected val liveItems: LiveData<List<Entity>> = roomDao.getItems()

val proxyItems = ArrayList<Proxy>()

protected abstract val data: Map<String, LiveProperty<out ItemEntity>>

abstract fun initData(db: StaffRoomDatabase): Map<String, LiveProperty<out ItemEntity>>

@Synchronized
protected fun refreshProxy() {
    proxyItems.clear()

    liveItems.value?.forEachIndexed { index, item ->
        proxyItems.add(
            createProxyItem(item, index)
        )
    }
    Log.d(this::class.simpleName, "Proxy Items was changed (${proxyItems.size})")

    notifyAdapter()
}

var adapterObserver: (() -> Unit)? = null

open fun notifyAdapter() {
    adapterObserver?.invoke()
}

abstract fun createProxyItem(
    item: Entity,
```

```
        index: Int
    ): Proxy

    @Synchronized override suspend fun insert(item: Proxy) {
        val mediateItem = dto.asMediateItem(item)

firestore.collection(collection).add(dto.asFirestoreItem(mediateItem)).addOnCompleteListener {
            scope.launch(Dispatchers.IO) {
                refresh()
            }
        }
    }

    @Synchronized override suspend fun update(item: Proxy) {
        val mediateItem = dto.asMediateItem(item)
        checkId(mediateItem.id)
        firestore.collection(collection).document(mediateItem.id)
            .set(dto.asFirestoreItem(mediateItem))
        roomDao.update(dto.asRoomItem(mediateItem))
    }

    @Synchronized override suspend fun delete(item: Proxy) {
        val mediateItem = dto.asMediateItem(item)
        checkId(mediateItem.id)
        firestore.collection(collection).document(mediateItem.id).delete()
        roomDao.delete(dto.asRoomItem(mediateItem))
    }

    abstract fun addProxyObserver(lifecycleOwner: LifecycleOwner)
}
```

ДОДАТОК 3

Аналітична система вибору кваліфікаційних груп забезпечення дисциплін

ОПИС ПРОГРАМНОГО МОДУЛЯ

УКР.КПІім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б_13-2

Аркушів 10

Київ 2020

АНОТАЦІЯ

Метою роботи було створення інформаційно-аналітичної системи для вибору кваліфікаційних груп забезпечення викладачів.

Модуль був розроблений у середовищі Android Studio мовою програмування Kotlin із використанням технологій таких як Firebase Cloud Store та бібліотекою для роботи із SQLite - Room.

Даний модуль виконує головні функції роботи із отримання та передачею інформації.

ЗМІСТ

1. ЗАГАЛЬНІ ВІДОМОСТІ

Модулі були написані на мові програмування Kotlin, для реалізації безпеки авторизації було використано готове рішення Firebase Authentication. Додаток надає можливість виконувати підбір інформаційних списків викладачів для вказаного предмету, та проводити аналіз вибору кваліфікаційних груп відповідно до ліцензійних вимог. Спілкування з базами даних відбувається у за допомогою SQL запитів та шляхом обміну інформацією через json.

Реалізація системи використовує декілька джерел даних для своєї роботи, та відповідає вимогам побудови інтерфейсів та створення мобільних додатків з використанням сучасних технологій. Це означає, що в майбутньому, у разі зміни конфігурації таблиць бази даних чи додаванні нових, систему буде дуже легко адаптувати до змін.

Інтерфейс та архітектура веб-додатку роблять розроблену систему легкою в використанні.

2. ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Абстрактний клас **AbstractRepository** призначений для забезпечення обміну даних між моделями додатку та базою даних. Цей клас містить загальний функціонал для роботи з репозіторіями і має характерну поведінку.

Серед методів поведінки є: приватний метод **updateTable**, який здійснює транзакцію обновлення локальної бази даних над конкретною таблицею бази. Цей метод звертається до об'єкту доступу до бази даних (DTO) з проханням замінити поточні елементи таблиці іншими, переданими як аргументи функції. Модифікатор доступу **private** забороняє доступ до методів чи полів класу за межами даного класу. Іншими словами поля і методи з таким модифікатором можуть явно використовуватись лише всередині класу, в якому вони оголошені.

Наступним є закритий метод поведінки **getAllFirestoreItems**. Він приймає функцію зворотного виклику у якості аргументу та виконує витягування даних конкретної таблиці із централізованого сховища даних. У випадку, якщо операціє була успішною, метод виконує виклик функції над отриманими елементами, переданої у якості аргументу цього методу. В більшості випадків функцією зворотного виклику являється метод **updateTable**, хоча можливі й інші варіації. Єдиною вимогою для переданої функції є те що вона повинна приймати як аргумент список елементів того ж типу, що власне й повертає центральна база даних у разі успішної операції.

Внутрішній метод поведінки **refreshItems** призначений для явного оновлення елементів локального кешу системи певної категорії. Він делегує запити до центральної бази даних методу, описаному попереднім, а також створює і передає функцію зворотного виклику, якщо вона є. Ця функція використовує диспетчер вводу/виводу (IO, з англ. Input/Output), щоб забезпечити, що операція з базою даних із вставкою в базу даних відбувається на диспетчері IO. Перемикаючись на диспетчер вводу-виводу за допомогою withContext, ця функція є безпечною для виклику з будь-якого потоку,

включаючи головний потік. Саме в цьому диспетчері відбувається оновлення локальної бази даних за допомогою функції класу **updateTable**. Далі контекст переключаються на головний потік, щоб виконати дію передану у якості аргументу цього методу. Зазвичай це деякі дії, пов’язані з оновленням інтерфейсу користувача, тому вони повинні виконуватись на головному потоці.

Усі вищеописані методи виділені в окремий внутрішній клас, що має назву **FirestoreBridge**, оскільки вони відрізняються від усіх інших і виконують функції, пов’язані з хмарною базою даних Firestore.

Публічні методи **refresh** і **refreshAndExecuteOnMain** виконують однакову функцію, за єдиним виключенням, що останній приймає у якості аргументу дію, яка буде виконана на головному потоці після оновлення локального кешу даних.

Функція класу **checkId** виконує перевірку ідентифікатору, переданого у якості аргументу і призводить до аварійного завершення програми, якщо ідентифікатор являється значенням за замовчуванням. Таке не допустимо, оскільки означає, що ідентифікатор для конкретного елементу не був встановлений. Це свідчить про помилку програми або про спробу зовнішнього втручання з ціллю заміни даних. В такому випадку, програму потрібно примусово зупинити.

Абстрактні методи класу **insert**, **update** та **delete** передбачають, що їх реалізація буде описана у класах-наслідниках. Такий підхід дозволяє викликати ці методи без інформації про їх реалізацію та працювати з класами наслідниками через інтерфейс базового класу.

Серед властивостей класу **AbstractRepository** можна виділити: назву конкретної таблиці у локальній БД, з якою працює поточний репозіторій, інтерфейс для взаємодії з центральною БД та композиція об’єкту класу **FirestoreBridge**.

Наступний абстрактний клас **ItemRepository** розширює функціонал класу **AbstractRepository** і є його прямим наслідником. Даний клас охоплює спільний функціонал для всіх репозиторіїв, що призначені для роботи з окремими компонентами системи. Він містить дві додаткових властивості, а саме: об’єкт доступу до даних (DAO) конкретної таблиці у базі та об’єкт перетворення даних (DTO) елементу відповідної таблиці. Останній забезпечує перетворення програмної структури у різні сутності, з якими можуть працювати або функції локальної БД, або центральної, а такою дозволяє перетворювати кожен з таких об’єктів у нейтральну сутність і навпаки.

Цей клас реалізує і описує абстрактні методи класу-предку.

Публічний метод **insert** приймає на вхід об’єкт отриманий шляхом перетворення запису з локально бази даних завдяки DAO. Він призначений для додавання нових записів як до центральної бази даних, так і до локальної. Спочатку відбувається вставка елементу до центрального сховища Firestore, а потім, у разі успіху, відбувається оновлення локальної бази даних через диспетчер вводу-виводу.

Реалізація методів **update** та **delete** мають аналогічний функціонал, за винятком того, що перший використовується для оновлення вже існуючого елементу, а останній – для видалення.

Наступний абстрактний клас **CombinedRepository** і черговий наслідник від **AbstractRepository** має схоже призначення, що й попередній клас. Різницею являється те, що поточний шаблон репозіторія працює зі складними об’єктами баз даних, в той час як **ItemRepository** – з простими. Під складними об’єктами бази даних мається на увазі те, що вони, на відміну від інших, мають залежності з іншими таблицями бази. А це означає, що такі елементи повинні оновлюватись на екрані користувача навіть тоді, коли зміни відбулись в елементах, на які посилається поточний об’єкт, навіть якщо останній не зазнав ніяких змін. Це є важливою задачею, оскільки елементи в базі, не знають нічого про стан елементів, на які вони посилаються. Наприклад, користувачу

не потрібно знати ідентифікатор навчальної дисципліни в базі, до якої відносить конкретний навчальний матеріал. В даному випадку користувача цікавить лише назва такої дисципліни. Тому якщо, наприклад назва такої дисципліни змінилась, інформація не екрані також повинна відображати такі зміни. Для такої цілі був створений допоміжний клас **LiveProperty**, який відслідковує подібні зміни та повідомляє про це назначеному слухачу.

Також окрім типів даних для локальної та центральної баз даних для кожної категорії або таблиці був створений інший тип даних з приставкою **Proxy**. Його роль в тому, щоб замінити отримані з бази даних ідентифікатори-посилання на набір інформації, яка потрібна користувачу про конкретний об’єкт за допомогою вказаного вище **LiveProperty**. Таким чином **CombinedRepository** описую поведінку роботи таких елементів. Серед специфічних методів варто виділити наступні:

Метод **refreshProxy** відповідає за оновлення проксі-даних. Це метод викликається коли надходить інформація, що відбулись зміни у самому об’єкті чи елементі, на який цей об’єкт посилається. Тут відбувається оновлення всіх змінених об’єктів, після чого посилається сигнал компоненту, відповідальному за оновлення користувацького інтерфейсу.

Поле **adapterObserver** містить у собі функцію, яка зберігаю інформація про те, як і кому потрібно передати сигнал про те, що відбулось оновлення елементів. Це поле приводиться у дію за допомогою функції класу **notifyAdapter**.

Методи **insert**, **delete** і **update** виконують ті ж самі функції, що й аналогічні у класі **ItemRepository**, за винятком того, що вони приймають проксі-об’єкт у якості аргументу функції і виконують характерні для нього перетворення.

3. ТЕХНІЧНІ ЗАСОБИ ЩО ВИКОРИСТОВУЮТЬСЯ

Даний програмний модуль розроблено у середовищі Android Studio розробником якої є компанії JetBrains та Google. Були використані такі мови програмування як Java, Kotlin та такі бібліотеки як Room, LiveData, ViewModel, Kotlinx та інші.

Програмний модуль було протестовано на персональному комп'ютері, який працює на базі процесору Intel Core i7-8750H 2.20GHz та має 16 Гб оперативної пам'яті на операційній системі Windows з використанням емулятора. Також тестування виконувалось на різних мобільних пристроях та планшетах, серед них: Google Pixel 2, Nexus 5X, Google Pixel 3A, Xiaomi Redmi Note 4X та інші.

Розроблене програмне забезпечення не залежить від роздільної здатності, щільності рідкокристалічного екрану чи розмірів девайсу та поставляється у вигляді .apk файлу. Це означає, що рішення дуже легко і просто встановити та почати користуватися.

4. ВИКЛИК І ЗАВАНТАЖЕННЯ

Для функціонування додатку необхідно:

1. Версія операційної системи Android 7.0 (Nougat) або вище
2. Завантажити мобільний додаток.
3. Встановити його.
4. Переконатись у наявності підключення до мережі Інтернет.

ДОДАТОК 4

Аналітична система вибору кваліфікаційних груп забезпечення дисциплін

Публікація тез XVIII Міжнародної
науково-практичної конференції
молодих вчених і студентів
2020 року

Том 2

УКР.КПІім.ІгоряСікорського_ТЕФ_АПЕПС_ТВ6137_20Б

Аркушів 4

Київ 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ПРОБЛЕМИ НАУКОВОГО ЗАБЕЗПЕЧЕННЯ ЕНЕРГЕТИКИ

Матеріали XVIII Міжнародної науково-практичної конференції молодих вчених і студентів 2020 року

ТОМ 2

Київ- 2020

УДК 004.054

Студент 4 курсу, гр. ТВ-61 Лавренюк В.В.
Ст.викл. Дацюк О.А.

ДЕКОМПОЗИЦІЯ ДАНИХ ПРИ МОДЕЛЮВАННІ ІНФОРМАЦІЙНО-ДОВІДКОВОЇ СИСТЕМИ

Щодня більшість людей проводять більше 5 годин за своїми смартфонами чи іншими мобільними пристроями. А це означає, що мобільні додатки визначають, як клієнти взаємодіють з вашим брендом. За даними Google, майже 9 з 10 людей порекомендують бренд після позитивного досвіду роботи з брендом на мобільних пристроях [1]. Отже при розробці програми реферального маркетингу, варто переконатися що вона оптимізована для мобільних пристроїв.

Google також зазначають, що "78% користувачів смартфонів частіше купують товари у компаній, що мають мобільні сайти чи додатки, які допомагають їм легко знайти відповіді на свої питання"[1]. Іншими словами, успішність вашого бренда напряму залежить від зручності інтерфейсу користувача.

Декомпозиція тут грає важливу роль, адже перенасиченість сторінки інформацією, з одного боку, є недостатньо зрозумілою для користувача, а з іншого – значно ускладнює подальшу розробку і масштабування системи.

Замість того щоб робити універсальну структуру, кращим рішенням є розподіл інформації та процесів, які з нею пов’язані, на дрібніші фрагменти. Кожен з таких фрагментів має відповідати на одне питання, залишаючись таким чином максимально простим у користуванні.

Можна сказати, що декомпозиція – це низхідний процес, що отримує породжуючу надмірність варіантів, і дає можливість ефективно перенастроювати первісну структуру відповідно до нових вимог [2]. Декомпозиція сприймається тут, як дія. Таким чином відбувається самоорганізація, як спосіб відтворення даної системи.

Якщо при декомпозиції з'ясовується, що модель починає описувати внутрішній алгоритм функціонування елемента замість закону його функціонування у вигляді «чорного ящика», то в цьому випадку відбулася зміна рівня абстракції. Це означає вихід за межі мети дослідження системи і, отже, викликає припинення декомпозиції.

Найчастіше декомпозиція проводиться шляхом побудови дерева цілей і дерева функцій. Таким чином, система може бути відображена у вигляді ієрархічної структури. Це справедливо не тільки для інформаційно-довідкової системи, а для системи в цілому.

Такий підхід до побудови алгоритмів роботи з системами впливає на проектування інтерфейсів роботи користувача. Метод декомпозиції використовує структуру завдання і дозволяє замінити рішення однієї великої задачі рішенням серії менших завдань. Хоча для більшості випадків достатньо буде реалізувати фільтри інформації внутрішньої структури за різними групами. А потім відображати потрібні дані порціями, коли в них з’являється необхідність.

Декомпозиція даних для програми реферального маркетингу або, іншими словами, інформаційно-довідкової системи дозволяє враховувати особливості функціонування систем та дає можливість контролювати роботу користувача з системою

Перелік посилань:

1. How mobile brand experiences help (or hurt) business results [Електронний ресурс] // Think with google. – 2017. – Режим доступу: https://www.thinkwithgoogle.com/data-collections/consumer-mobile-brand-experiences/.

2. Проблема декомпозиції в математичному моделюванні / Ю. Н. Павловский, Т. Г. Смирнова. — М.: ФАЗИС, 1998. ISBN 5-7036-0046-4